華中科技大學

操作系统原理课程设计报告

姓　　名：	刘本嵩
学　　院：	计算机科学与技术
专　　业：	计算机科学与技术
班　　级：	CS1601
学　　号：	U201614531
指导教师：	阳富民

分数	
教师签名	

2020 年 5 月 13 日

目　录

1 实验一 Linux 用户界面的使用

1.1 实验目的

掌握 Linux 操作系统的使用方法；

了解 Linux 系统内核代码结构；

掌握实例操作系统的实现方法。

1.2 实验内容

1. 编一个 C 程序，其内容为实现文件拷贝的功能。基本要求：使用系统调用 open/read/write...；

2. 编一个 C 程序，其内容为分窗口同时显示三个并发进程的运行结果。要求用到 Linux 下的图形库。 (gtk/Qt)三个独立子进程，各自窗口显示；三个进程誊抄演示。

1.3 实验设计

1.3.1 开发环境

```
recolic@RECOLICMPC
OS: Manjaro 20.0 Lysia
Kernel: x86_64 Linux 4.19.120-1-MANJARO
Uptime: 21m
Packages: 1852
Shell: fish 3.1.1
Resolution: 1920x1080
DE: GNOME 3.36.2
WM: Mutter
WM Theme:
```

```
GTK Theme: Adwaita [GTK2/3]
Icon Theme: Adwaita
Font: Cantarell 11
Disk: 65G / 111G (62%)
CPU: Intel Core m3-7Y30 @ 4x 2.6GHz [61.0°C]
GPU: Intel Corporation HD Graphics 615 (rev 02)
RAM: 2968MiB / 3827MiB
```

需要注意的是，本次实验的所有 C++代码，均依赖于我自己实现的 rlib 库 (https://github.com/recolic/rlib). 此程序库所有代码均由我原创实现.

此实验中均使用 gcc 9.3.0 的 C++2a 标准.

1.3.2 实验设计

注意到实验内容 1 和实验内容 2 的相似性，我们实现如下两个程序:

1. cp 程序. 其接受--gui 参数，可以选择开启或关闭 GUI 功能.
2. 誊抄程序. 它调用 3 次上面的 cp 程序，并带上--gui 参数. 并且创建临时的命名管道，用来作为 3 个 cp 程序的中间管道.

我们首先实现一个简单的 GUI，这里直接调用 gtkmm 的库来创建和更新窗口，具体代码如下:

```
#include <gtkmm.h>

class ProcGUI : public Gtk::Window
{
public:
	ProcGUI(std::string title, std::string initTxt)
		: m_txt(initTxt.c_str()), copiedBytes(0), finished(false) {
		set_title(title.c_str());
		set_border_width(10);
		dispatcher.connect(sigc::mem_fun(*this,
&ProcGUI::dispatcherHandler));
		add(m_txt);
		m_txt.show();
	}
	virtual ~ProcGUI() {}
```

```
    std::atomic<size_t> copiedBytes;
    std::atomic<bool> finished;
    Glib::Dispatcher dispatcher;
protected:
    Gtk::Label m_txt;
    void dispatcherHandler() {
        auto prefix = finished ? "Finished: " : "Copied: ";
        std::string s = std::string() + prefix +
fsizeToString(copiedBytes) + "B";
        m_txt.set_text(s.c_str());
    }
};
```

随后，我们对 cp 程序的主要拷贝逻辑进行实现．如果目标文件不存在，它使用源文件相同的权限位来创建目标文件并开始拷贝．在循环中，如果检测到 GUI 窗口存在，则会实时更新 GUI 窗口的内容，显示拷贝的实时进度．

```
void do_copy(string src, string dst, ProcGUI *guiPtr) {
    auto srcFd = open(src.c_str(), O_RDONLY);
    auto dstFd = creat(dst.c_str(), get_file_permission(src));
    if(srcFd == -1) throw std::runtime_error("Unable to open {} for
read, {}"_rs.format(src, strerror(errno)));
    rlib_defer([&]{ close(srcFd); });
    if(dstFd == -1) throw std::runtime_error("Unable to open {} for
write, {}"_rs.format(src, strerror(errno)));
    rlib_defer([&]{ close(dstFd); });

    constexpr size_t buf_size = 4096; // 64K
    char buf[buf_size];
    time_t last_update_time = time(NULL);
    while(true) {
        auto size = read(srcFd, buf, buf_size);
        if(size == -1) throw std::runtime_error("read error");
        if(size == 0) break; // EOF
        rlib::fdIO::writen_ex(dstFd, buf, size);
        if(guiPtr) {
            guiPtr->copiedBytes += size;
            if(auto curr_time = time(NULL); curr_time >
last_update_time) {
                last_update_time = curr_time;
                guiPtr->dispatcher.emit(); // Only once per second.
            }
        }
    }

    if(guiPtr) {
        guiPtr->finished = true;
        guiPtr->dispatcher.emit();
    }
}
```

最后，为 cp 程序提供一个 main 函数，用来处理输入的命令行参数. 如果需要显示 GUI，就创建一个 ProcGUI 实例，将参数中的 title 传入即可.

```
int main(int argc, char **argv) {
    opt_parser args(argc, argv);
    bool guiMode = args.getBoolArg("--gui");
    auto windowTitle = args.getValueArg("--title", false, args.getSelf());
    if(args.data().size() != 2)
        throw std::runtime_error("Copies two file stream.\nUsage: cp $srcFname $dstFname\nOptions: [--gui] [--title $title]");
    auto src = args.data()[0];
    auto dst = args.data()[1];

    // GUI business.
    auto app = Gtk::Application::create(); // Requires GTKmm 3.6+ to set empty application id. See Ref.
    ProcGUI procGUI(windowTitle, "Copied: 0B");
    if(guiMode) {
        std::thread(&do_copy, src, dst, &procGUI).detach();
        app->run(procGUI);
    }
    else {
        do_copy(src, dst, nullptr);
    }
}
```

在完成了以上的 cp 程序后，第二个程序就显得很简单了. 首先生成 3 个命名管道作为中间文件名，然后 fork 出 3 个进程，利用 rlib 的 execs 函数，将 cp 程序用正确的参数执行 3 遍即可. 值得注意的是，需要恰当的设置 GUI 窗口的标题. 具体代码如下所示:

```
int main(int argc, char **argv) {
    // INPUT --> PIPE1 --> PIPE2 --> OUTPUT
    //        ProcA       ProcB       ProcC
    rlib::opt_parser args(argc, argv);
    if(args.data().size() != 2)
        throw std::runtime_error("Copies two file stream.\nUsage:
game $srcFname $dstFname");
    auto src = args.data()[0];
    auto dst = args.data()[1];

    auto pipe1 = "/tmp/recolic-hust-os-fifo-518922714", pipe2 =
"/tmp/recolic-hust-os-fifo-125350723";
    remove(pipe1); remove(pipe2); // Just make a try. No error check.
    if(mkfifo(pipe1, 0666) == -1 || mkfifo(pipe2, 0666) == -1)
        throw std::runtime_error("mkfifo");

    auto pids = std::make_pair(fork(), fork());
    if(pids.first == -1 || pids.second == -1)
        throw std::runtime_error("fork");
    rlib::println(pids.first, pids.second);

    if(pids.first == 0 && pids.second == 0)
        exit(0); // Too many processes...
    if(pids.first == 0) {
        // Proc A
        rlib::execs("./cp", std::vector<std::string>{src, pipe1, "--gui",
"--title", "A"});
    }
    else if(pids.second == 0) {
        // Proc B
        rlib::execs("./cp", std::vector<std::string>{pipe1, pipe2,
"--gui", "--title", "B"});
    }
    else {
        // Proc C
        rlib::execs("./cp", std::vector<std::string>{pipe2, dst, "--gui",
"--title", "C"});
    }
    throw std::runtime_error("execl returns.");
}
```

1.4 实验调试

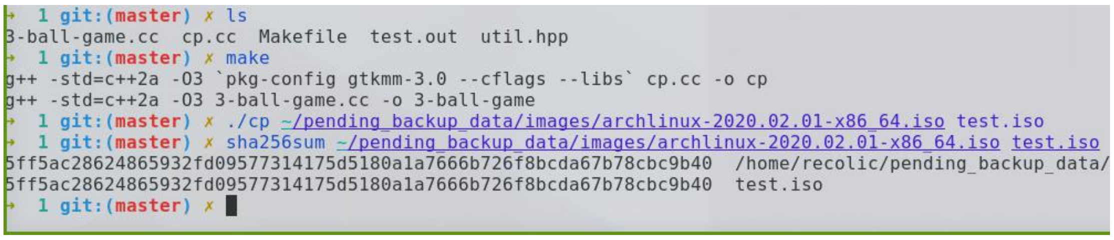
1.4.1 实验步骤

首先对代码结构进行大致设计，对 GUI，拷贝逻辑这两个主要模块进行实现. 随后，为 cp 程序实现 main 函数. 随后进行 cp 程序的测试.

在 cp 程序测试通过后，3-ball-game 程序通过 fork 和 execs 来调用 3 次 cp 程序，并启用 cp 程序的 GUI 功能. 最后，使用大文件对这两个程序进行拷贝测试，表明程序工作正常.

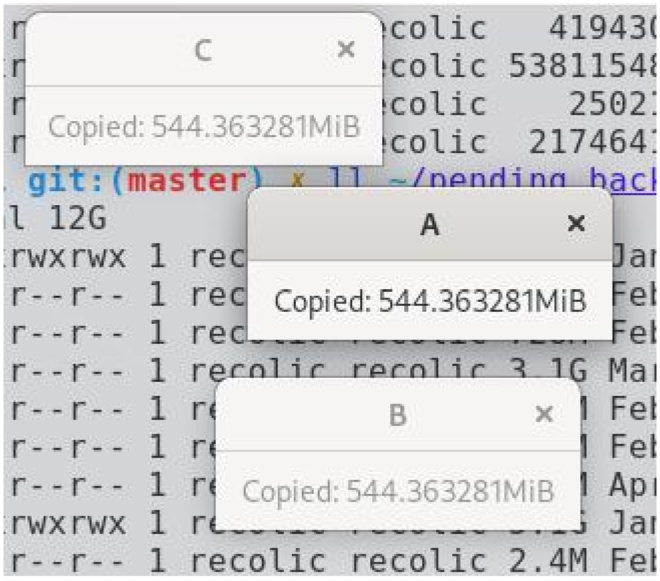
1.4.2 实验调试及心得

首先对 cp 程序进行测试，对 archlinux 2020 年 2 月的镜像文件进行拷贝，文件大小约 650MB. 测试表明拷贝在正常速度下完成，速度与 linux 自带 cp 程序相当，文件内容正确. 测试过程如下图所示.



```
→  1 git:(master) ✗ ls
3-ball-game.cc  cp.cc  Makefile  test.out  util.hpp
→  1 git:(master) ✗ make
g++ -std=c++2a -O3 `pkg-config gtkmm-3.0 --cflags --libs` cp.cc -o cp
g++ -std=c++2a -O3 3-ball-game.cc -o 3-ball-game
→  1 git:(master) ✗ ./cp ~/pending_backup_data/images/archlinux-2020.02.01-x86_64.iso test.iso
→  1 git:(master) ✗ sha256sum ~/pending_backup_data/images/archlinux-2020.02.01-x86_64.iso test.iso
5ff5ac28624865932fd09577314175d5180a1a7666b726f8bcda67b78cbc9b40  /home/recolic/pending_backup_data/
5ff5ac28624865932fd09577314175d5180a1a7666b726f8bcda67b78cbc9b40  test.iso
→  1 git:(master) ✗ █
```

然后对 3 进程誊抄(3-ball-game)程序进行测试. 同样进行编译，使用 archlinux 镜像文件进行测试，拷贝过程中能够正常显示 3 个窗口，每个窗口都能正确显示当前拷贝的进度. 程序界面如下图所示.

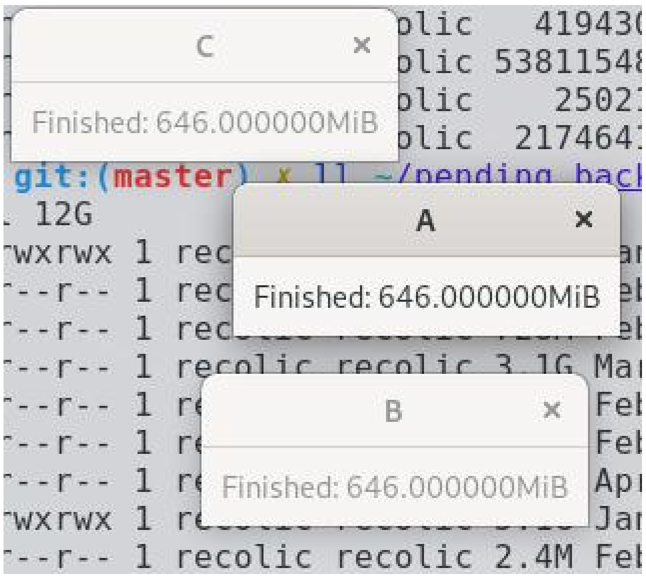


```
C
Copied: 544.363281MiB

A
Copied: 544.363281MiB

B
Copied: 544.363281MiB
```

拷贝完成后，程序从显示实时进度，改为显示 Finished.

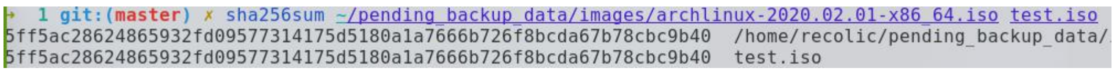


```
C
Finished: 646.000000MiB

A
Finished: 646.000000MiB

B
Finished: 646.000000MiB
```

同样对拷贝好的目标文件的 sha256 校验和进行计算，结果表明文件内容正确.



```
→ 1 git:(master) ✗ sha256sum ~/pending_backup_data/images/archlinux-2020.02.01-x86_64.iso test.iso
5ff5ac28624865932fd09577314175d5180a1a7666b726f8bcda67b78cbc9b40  /home/recolic/pending_backup_data/
5ff5ac28624865932fd09577314175d5180a1a7666b726f8bcda67b78cbc9b40  test.iso
```

在本此实验中，我对程序的各个模块进行了较细致的划分，在后来的 3 进程誊抄程序中很好的复用了前面的 cp 程序中的代码. 这种模块化的设计方法，为后面程序的设计减少了工作量，也增加了代码的可读性和可维护性，减少了出问题的概率.

附录 实验代码

```
//# ----- 3-ball-game.cc
#include <utility>
#include <stdlib.h>
#include <unistd.h>
#include <rlib/stdio.hpp>
#include <rlib/opt.hpp>
#include <rlib/sys/unix_handy.hpp>

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

int main(int argc, char **argv) {
    // INPUT --> PIPE1 --> PIPE2 --> OUTPUT
    //        ProcA      ProcB      ProcC
    rlib::opt_parser args(argc, argv);
    if(args.data().size() != 2)
        throw std::runtime_error("Copies two file stream.\nUsage:
game $srcFname $dstFname");
    auto src = args.data()[0];
    auto dst = args.data()[1];

    auto pipe1 = "/tmp/recolic-hust-os-fifo-518922714", pipe2 =
"/tmp/recolic-hust-os-fifo-125350723";
    remove(pipe1); remove(pipe2); // Just make a try. No error check.
    if(mkfifo(pipe1, 0666) == -1 || mkfifo(pipe2, 0666) == -1)
        throw std::runtime_error("mkfifo");

    auto pids = std::make_pair(fork(), fork());
    if(pids.first == -1 || pids.second == -1)
        throw std::runtime_error("fork");
    rlib::println(pids.first, pids.second);

    if(pids.first == 0 && pids.second == 0)
        exit(0); // Too many processes...
    if(pids.first == 0) {
        // Proc A
        rlib::execs("./cp", std::vector<std::string>{src, pipe1, "--gui",
"--title", "A"});
    }
    else if(pids.second == 0) {
        // Proc B
        rlib::execs("./cp",    std::vector<std::string>{pipe1,    pipe2,
"--gui", "--title", "B"});
    }
    else {
        // Proc C
        rlib::execs("./cp", std::vector<std::string>{pipe2, dst, "--gui",
"--title", "C"});
    }
    throw std::runtime_error("execl returns.");
}

//# ----- cp.cc
#include <rlib/sys/sio.hpp>
#include <rlib/stdio.hpp>
#include <rlib/opt.hpp>
#include <rlib/string.hpp>
#include <thread>
#include "util.hpp"

using namespace rlib;
using namespace rlib::literals;

void do_copy(string src, string dst, ProcGUI *guiPtr);

int main(int argc, char **argv) {
    opt_parser args(argc, argv);
    bool guiMode = args.getBoolArg("--gui");
    auto    windowTitle    =    args.getValueArg("--title",    false,
args.getSelf());
    if(args.data().size() != 2)
        throw std::runtime_error("Copies two file stream.\nUsage: cp
$srcFname $dstFname\nOptions: [--gui] [--title $title]");
    auto src = args.data()[0];
    auto dst = args.data()[1];

    // GUI business.
    auto app = Gtk::Application::create(); // Requires GTKmm 3.6+ to
set empty application id. See Ref.
    ProcGUI procGUI(windowTitle, "Copied: 0B");
    if(guiMode) {
        std::thread(&do_copy, src, dst, &procGUI).detach();
        app->run(procGUI);
    }
    else {
        do_copy(src, dst, nullptr);
    }
}

void do_copy(string src, string dst, ProcGUI *guiPtr) {
    auto srcFd = open(src.c_str(), O_RDONLY);
    auto dstFd = creat(dst.c_str(), get_file_permission(src));
    if(srcFd == -1) throw std::runtime_error("Unable to open {} for
read, {}"_rs.format(src, strerror(errno)));
    rlib_defer([&]{ close(srcFd); });
    if(dstFd == -1) throw std::runtime_error("Unable to open {} for
write, {}"_rs.format(src, strerror(errno)));
    rlib_defer([&]{ close(dstFd); });

    constexpr size_t buf_size = 4096; // 64K
    char buf[buf_size];
    time_t last_update_time = time(NULL);
    while(true) {
        auto size = read(srcFd, buf, buf_size);
        if(size == -1) throw std::runtime_error("read error");
        if(size == 0) break; // EOF
        rlib::fdIO::writen_ex(dstFd, buf, size);
        if(guiPtr) {
            guiPtr->copiedBytes += size;
            if(auto curr_time = time(NULL); curr_time >
last_update_time) {
                last_update_time = curr_time;
                guiPtr->dispatcher.emit(); // Only once per second.
            }
        }
    }

    if(guiPtr) {
        guiPtr->finished = true;
        guiPtr->dispatcher.emit();
    }
}



//# ----- Makefile

CXX ?= g++
CXXFLAGS = -std=c++2a -O3
```

```
def:
	$(CXX) $(CXXFLAGS) `pkg-config gtkmm-3.0 --cflags --libs` cp.cc
-o cp
	$(CXX) $(CXXFLAGS) 3-ball-game.cc -o 3-ball-game

clean:
	rm -f cp 3-ball-game


//# ----- util.hpp
#ifndef R_HUST_OS_DESIGN_UTIL_HPP_
#define R_HUST_OS_DESIGN_UTIL_HPP_ 1

#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <string>
#include <atomic>
#include <stdexcept>

inline mode_t get_file_permission(std::string fname) {
	struct stat statbuf;
	auto res = stat(fname.c_str(), &statbuf);
	if(res == -1)
		throw std::runtime_error(std::string("Unable to stat file ") +
fname);
	return statbuf.st_mode;
}

//                              Copied                              from
https://github.com/recolic/hust-os-exp/blob/master/fs_prettyprint.hp
p
inline std::string fsizeToString(const size_t fsize) {
	if(fsize < 1024)
		return std::to_string(fsize);
	const auto KiB = (double)fsize / 1024.;
	if(KiB < 1024) return std::to_string(KiB) + "Ki";
	const auto MiB = KiB / 1024.;
	if(MiB < 1024) return std::to_string(MiB) + "Mi";
	const auto GiB = MiB / 1024.;
	if(GiB < 1024) return std::to_string(GiB) + "Gi";
	const auto TiB = GiB / 1024.;
	if(TiB < 1024) return std::to_string(TiB) + "Ti";
	const auto PiB = TiB / 1024.;
	return std::to_string(PiB) + "Pi";
}

#include <gtkmm.h>

class ProcGUI : public Gtk::Window
{
public:
	ProcGUI(std::string title, std::string initTxt)
		: m_txt(initTxt.c_str()), copiedBytes(0), finished(false) {
		set_title(title.c_str());
		set_border_width(10);
		dispatcher.connect(sigc::mem_fun(*this,
&ProcGUI::dispatcherHandler));
		add(m_txt);
		m_txt.show();
	}
	virtual ~ProcGUI() {}

	std::atomic<size_t> copiedBytes;
	std::atomic<bool> finished;
	Glib::Dispatcher dispatcher;
protected:
	Gtk::Label m_txt;
	void dispatcherHandler() {
		auto prefix = finished ? "Finished: " : "Copied: ";
		std::string	s	=	std::string()	+	prefix	+
fsizeToString(copiedBytes) + "B";
		m_txt.set_text(s.c_str());
	}
};



#endif
```

2 实验二 新增系统调用

2.1 实验目的

掌握 Linux 操作系统的使用方法；

了解 Linux 系统内核代码结构；

掌握实例操作系统的实现方法。

2.2 实验内容

1. 内核编译、生成，用新内核启动；

2. 新增系统调用实现：文件拷贝或 P、V 操作。

2.3 实验设计

2.3.1 开发环境

```
recolic@RECOLICMPC
OS: Manjaro 20.0 Lysia
Kernel: x86_64 Linux 4.19.120-1-MANJARO
Uptime: 21m
Packages: 1852
Shell: fish 3.1.1
Resolution: 1920x1080
DE: GNOME 3.36.2
WM: Mutter
WM Theme:
GTK Theme: Adwaita [GTK2/3]
Icon Theme: Adwaita
Font: Cantarell 11
Disk: 65G / 111G (62%)
CPU: Intel Core m3-7Y30 @ 4x 2.6GHz [61.0°C]
```

GPU: Intel Corporation HD Graphics 615 (rev 02)
RAM: 2968MiB / 3827MiB

本次实验开发过程中使用 Linux5.5 版本的源代码.

需要注意的是，本次实验的所有 C++代码，均依赖于我自己实现的 rlib 库 (https://github.com/recolic/rlib). 此程序库所有代码均由我原创实现.

此实验中均使用 gcc 9.3.0 的 C++2a 标准.

2.3.2 实验设计

本次实验较为简单，只需要阅读内核开发组提供的，对应版本的内核文档 (https://www.kernel.org/doc/html/v5.5/process/adding-syscalls.html)，并且按照文档的说明一步一步进行系统调用的增加即可.

根据内核开发者的文档，应当进行以下几个步骤:

1. 修改 syscall_32.tbl 和 syscall_64.tbl，将新的 syscall 条目加入到列表中.
2. 修改 syscalls/linux.h，增加新的 asm linkage 开头的 syscall 函数声明.
3. 修改 uapi/asm-generic/unistd.h，利用__SYSCALL 宏，增加你的 syscall.
4. 修改 sys_ni.c，利用 COND_SYSCALL 宏，将新的 syscall 加入列表.
5. 增加新的源文件/homework/hust_cp.c 和对应的 Makefile, test.c，同时修改/Makefile，将新的 homework 目录加入 Makefile 编译 targets(core-y)中.

2.4 实验调试

2.4.1 实验步骤

首先按照 2.3.2 节的实验设计，完成内核开发文档要求的步骤. 随后实现 hust_cp.c，使用 SYSCALL_DEFINE3 宏，增加系统调用 hust_cp. 同时增加两

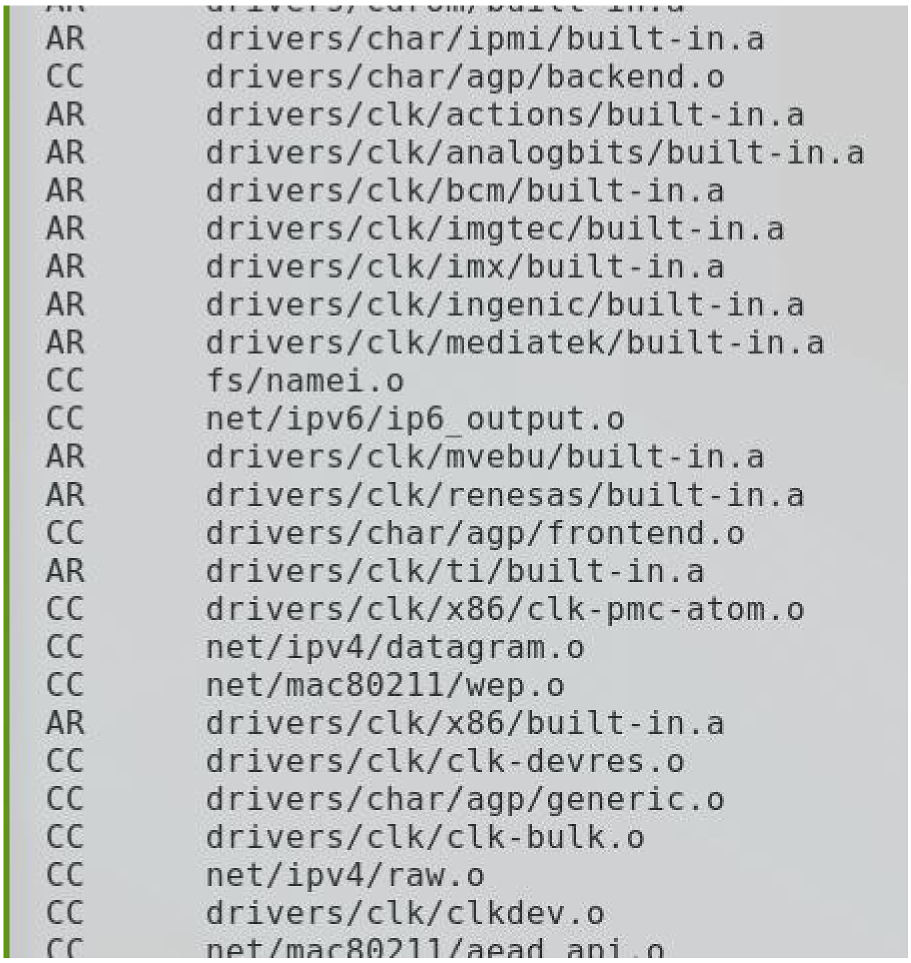
个有用的内核态辅助函数, kernel_read_by_fd 和 kernel_write_by_fd. 这个系统调用接受 2 个用户态的字符串, srcname 和 dstname. 接受一个整数，表示目的文件打开时的模式.

hust_cp 这个系统调用，使用一个类似于实验一中 cp 程序的循环，调用 ksys_open, ksys_close, kernel_read_by_fd 和 kernel_write_by_fd 进行操作. 在 kernel_read_by_fd 和 write 中，调用 kernel_read 和 kernel_write 最终完成读写.

随后保存代码，编译内核，启动进入系统. 随后编译 test.c 进行测试.

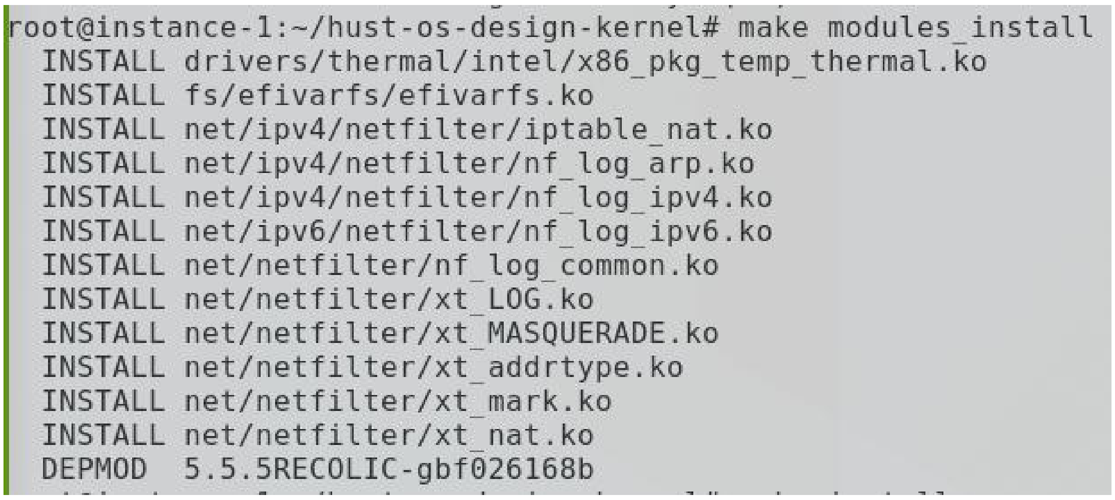
2.4.2 实验调试及心得

首先编译内核代码，使用 Manjaro 自带的内核配置文件作为.config 文件，在服务器上运行 make -j64 对内核进行编译.



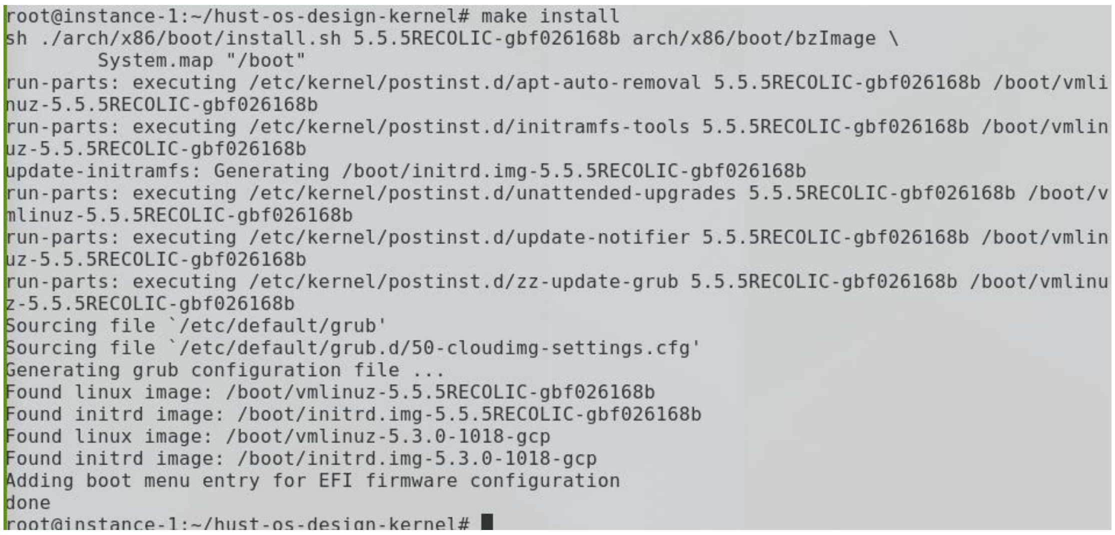
```
  AR      drivers/char/ipmi/built-in.a
  CC      drivers/char/agp/backend.o
  AR      drivers/clk/actions/built-in.a
  AR      drivers/clk/analogbits/built-in.a
  AR      drivers/clk/bcm/built-in.a
  AR      drivers/clk/imgtec/built-in.a
  AR      drivers/clk/imx/built-in.a
  AR      drivers/clk/ingenic/built-in.a
  AR      drivers/clk/mediatek/built-in.a
  CC      fs/namei.o
  CC      net/ipv6/ip6_output.o
  AR      drivers/clk/mvebu/built-in.a
  AR      drivers/clk/renesas/built-in.a
  CC      drivers/char/agp/frontend.o
  AR      drivers/clk/ti/built-in.a
  CC      drivers/clk/x86/clk-pmc-atom.o
  CC      net/ipv4/datagram.o
  CC      net/mac80211/wep.o
  AR      drivers/clk/x86/built-in.a
  CC      drivers/clk/clk-devres.o
  CC      drivers/char/agp/generic.o
  CC      drivers/clk/clk-bulk.o
  CC      net/ipv4/raw.o
  CC      drivers/clk/clkdev.o
  CC      net/mac80211/aead_api.o
```

等待编译完成后，运行 make modules_install 对 modules 进行编译和安装.



```
root@instance-1:~/hust-os-design-kernel# make modules_install
  INSTALL drivers/thermal/intel/x86_pkg_temp_thermal.ko
  INSTALL fs/efivarfs/efivarfs.ko
  INSTALL net/ipv4/netfilter/iptable_nat.ko
  INSTALL net/ipv4/netfilter/nf_log_arp.ko
  INSTALL net/ipv4/netfilter/nf_log_ipv4.ko
  INSTALL net/ipv6/netfilter/nf_log_ipv6.ko
  INSTALL net/netfilter/nf_log_common.ko
  INSTALL net/netfilter/xt_LOG.ko
  INSTALL net/netfilter/xt_MASQUERADE.ko
  INSTALL net/netfilter/xt_addrtype.ko
  INSTALL net/netfilter/xt_mark.ko
  INSTALL net/netfilter/xt_nat.ko
  DEPMOD  5.5.5RECOLIC-gbf026168b
```

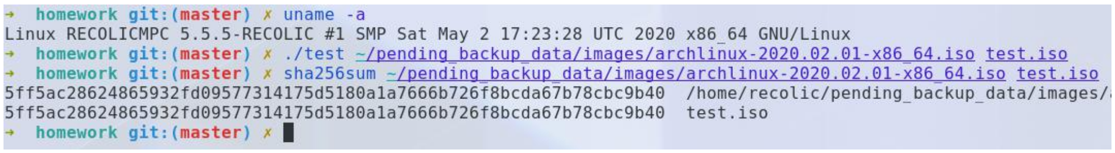
最后，使用 make install 将内核安装到服务器系统中.



```
root@instance-1:~/hust-os-design-kernel# make install
sh ./arch/x86/boot/install.sh 5.5.5RECOLIC-gbf026168b arch/x86/boot/bzImage \
        System.map "/boot"
run-parts: executing /etc/kernel/postinst.d/apt-auto-removal 5.5.5RECOLIC-gbf026168b /boot/vmli
nuz-5.5.5RECOLIC-gbf026168b
run-parts: executing /etc/kernel/postinst.d/initramfs-tools 5.5.5RECOLIC-gbf026168b /boot/vmlin
uz-5.5.5RECOLIC-gbf026168b
update-initramfs: Generating /boot/initrd.img-5.5.5RECOLIC-gbf026168b
run-parts: executing /etc/kernel/postinst.d/unattended-upgrades 5.5.5RECOLIC-gbf026168b /boot/v
mlinuz-5.5.5RECOLIC-gbf026168b
run-parts: executing /etc/kernel/postinst.d/update-notifier 5.5.5RECOLIC-gbf026168b /boot/vmlin
uz-5.5.5RECOLIC-gbf026168b
run-parts: executing /etc/kernel/postinst.d/zz-update-grub 5.5.5RECOLIC-gbf026168b /boot/vmlinu
z-5.5.5RECOLIC-gbf026168b
Sourcing file `/etc/default/grub'
Sourcing file `/etc/default/grub.d/50-cloudimg-settings.cfg'
Generating grub configuration file ...
Found linux image: /boot/vmlinuz-5.5.5RECOLIC-gbf026168b
Found initrd image: /boot/initrd.img-5.5.5RECOLIC-gbf026168b
Found linux image: /boot/vmlinuz-5.3.0-1018-gcp
Found initrd image: /boot/initrd.img-5.3.0-1018-gcp
Adding boot menu entry for EFI firmware configuration
done
root@instance-1:~/hust-os-design-kernel# █
```

值得注意的是，在某些没有 LILO 的发行版中(例如 Manjaro)，只需将编译好的 bzImage 复制到/boot/vmlinuz-x.x-x86_64，并运行 grub-mkconfig 和 mkinitcpio，即可手动完成 install 的安装过程．此处不在赘述.

在安装成功后，重启进入到新的内核中．在新的内核下编译 homework/test.cc 并运行，可以观察到，系统已经在使用新的内核，并且通过系统调用完成文件拷贝的过程可以正确完成，此测试过程如下图所示.



```
→  homework git:(master) ✗ uname -a
Linux RECOLICMPC 5.5.5-RECOLIC #1 SMP Sat May 2 17:23:28 UTC 2020 x86_64 GNU/Linux
→  homework git:(master) ✗ ./test ~/pending_backup_data/images/archlinux-2020.02.01-x86_64.iso test.iso
→  homework git:(master) ✗ sha256sum ~/pending_backup_data/images/archlinux-2020.02.01-x86_64.iso test.iso
5ff5ac28624865932fd09577314175d5180a1a7666b726f8bcda67b78cbc9b40  /home/recolic/pending_backup_data/images/
5ff5ac28624865932fd09577314175d5180a1a7666b726f8bcda67b78cbc9b40  test.iso
→  homework git:(master) ✗ █
```

在本次实验中，最重要的工作就是正确的阅读内核开发者提供的文档，并根据文档要求正确实现内核态的函数.

附录 实验代码

由于本次实验是对 Linux 内核的修改，因此实验代码使用 git diff 格式给出．原内核代码树位置为 Linux5.5 的 release 包.

```
diff --git a/Makefile b/Makefile
index 1f7dc3a2e..c1f5c4785 100644
--- a/Makefile
+++ b/Makefile
```

```
@@ -1014,7 +1014,7 @@ export MODORDER := $(extmod-prefix)modules.order
 export MODULES_NSDEPS := $(extmod-prefix)modules.nsdeps

 ifeq ($(KBUILD_EXTMOD),)
-core-y      += kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/
+core-y      += kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/ homework/

 vmlinux-dirs := $(patsubst %/,%,$(filter %/, $(init-y) $(init-m) \
              $(core-y) $(core-m) $(drivers-y) $(drivers-m) \
diff --git a/arch/x86/entry/syscalls/syscall_32.tbl b/arch/x86/entry/syscalls/syscall_32.tbl
index 15908eb9b..eed271a78 100644
--- a/arch/x86/entry/syscalls/syscall_32.tbl
+++ b/arch/x86/entry/syscalls/syscall_32.tbl
@@ -440,3 +440,4 @@
 433   i386   fspick          sys_fspick          __ia32_sys_fspick
 434   i386   pidfd_open      sys_pidfd_open      __ia32_sys_pidfd_open
 435   i386   clone3          sys_clone3          __ia32_sys_clone3
+436   i386   hust_cp         sys_hust_cp         __ia32_sys_hust_cp
diff --git a/arch/x86/entry/syscalls/syscall_64.tbl b/arch/x86/entry/syscalls/syscall_64.tbl
index c29976eca..1029532f7 100644
--- a/arch/x86/entry/syscalls/syscall_64.tbl
+++ b/arch/x86/entry/syscalls/syscall_64.tbl
@@ -357,6 +357,7 @@
 433   common   fspick          __x64_sys_fspick
 434   common   pidfd_open      __x64_sys_pidfd_open
 435   common   clone3          __x64_sys_clone3/ptregs
+436 common   hust_cp          __x64_sys_hust_cp

 #
 # x32-specific system call numbers start at 512 to avoid cache impact
diff --git a/homework/Makefile b/homework/Makefile
new file mode 100644
index 000000000..ab571e4ac
--- /dev/null
+++ b/homework/Makefile
@@ -0,0 +1,8 @@
+obj-y:=hust_cp.o
```

```
+
+recolic_test:
+	g++ test.cc -o test
+
+clean:
+	rm -f test
+
diff --git a/homework/hust_cp.c b/homework/hust_cp.c
new file mode 100644
index 000000000..b9b230bb6
--- /dev/null
+++ b/homework/hust_cp.c
@@ -0,0 +1,102 @@
+/* Syscall `hust_cp` as HUST homework, with syscall number 436,
+ * Copyright (C) Recolic Keghart <root@recolic.net>, 2020.
+ **/
+
+#include <linux/kernel.h>
+#include <linux/init.h>
+#include <linux/sched.h>
+#include <linux/syscalls.h>
+
+#define auto __auto_type
+
+static ssize_t kernel_read_by_fd(int fd, void *buf, size_t count, loff_t
*pos) {
+	struct fd f = fdget(fd);
+	ssize_t ret = -EBADF;
+
+	if (!f.file)
+		goto out;
+
+	ret = kernel_read(f.file, buf, count, pos);
+out:
+	fdput(f);
+	return ret;
+}
+
+static ssize_t kernel_write_by_fd(int fd, const void *buf, size_t count,
loff_t *pos) {
+	struct fd f = fdget(fd);
+	ssize_t ret = -EBADF;
+
+	if (!f.file)
```

```
+           goto out;
+
+      ret = kernel_write(f.file, buf, count, pos);
+out:
+      fdput(f);
+      return ret;
+}
+
+#define HUST_CP_BUF_SIZE 4096
+SYSCALL_DEFINE3(hust_cp, const char __user *, srcfname, const
char __user *, dstfname, umode_t, dst_mode) {
+      int ret;
+
+      long src_fd = ksys_open(srcfname, O_RDONLY, NULL);
+        long dst_fd = ksys_open(dstfname, O_CREAT | O_WRONLY |
O_TRUNC, dst_mode);
+      if(src_fd < 0 || dst_fd < 0) {
+           ret = src_fd < 0 ? src_fd : dst_fd;
+           goto out_without_close;
+      }
+
+      void *buf = vmalloc(HUST_CP_BUF_SIZE); // will be vmalloc-ed
+      if(!buf) {
+           ret = -ENOMEM;
+           goto out;
+      }
+
+      ssize_t actual_size;
+      loff_t src_pos = 0, dst_pos = 0;
+      while(true) {
+                      actual_size  =  kernel_read_by_fd(src_fd,  buf,
HUST_CP_BUF_SIZE, &src_pos);
+           if(actual_size < 0) {
+                ret = actual_size;
+                goto out;
+           }
+           if(actual_size == 0) {
+                break;
+           }
+
+             actual_size = kernel_write_by_fd(dst_fd, buf, actual_size,
&dst_pos);
+           if(actual_size < 0) {
+                ret = actual_size;
+                       goto out;
+               }
+       }
+
+out:
+       ksys_close(src_fd);
+       ksys_close(dst_fd);
+out_without_close:
+       if(buf)
+               vfree(buf);
+       return ret;
+
+}
+
diff --git a/homework/test.cc b/homework/test.cc
new file mode 100644
index 000000000..afda0050d
--- /dev/null
+++ b/homework/test.cc
@@ -0,0 +1,24 @@
+#include <sys/syscall.h>
+//#include <linux/kernel.h>
+#include <unistd.h>
+
+#include <rlib/opt.hpp>
+#include <rlib/stdio.hpp>
+#include <stdexcept>
+
+#define __NR_hust_cp 436
+
+int main(int argc, char **argv) {
+       rlib::opt_parser args(argc, argv);
+       if(args.data().size() != 2)
+                       throw std::runtime_error("Usage: ./this $srcFname
$dstFname");
+
+       const char *src = args.data()[0].c_str();
+       const char *dst = args.data()[1].c_str();
+
+       int ret = syscall(__NR_hust_cp, src, dst, 0644);
+       if(ret != 0)
+         rlib::println("Copy failed ({}), {}.", ret, strerror(errno));
+       return ret;
+}
```

```
+
diff --git a/include/linux/syscalls.h b/include/linux/syscalls.h
index 5262b7a76..3863fa55a 100644
--- a/include/linux/syscalls.h
+++ b/include/linux/syscalls.h
@@ -1001,6 +1001,12 @@ asmlinkage long
sys_pidfd_send_signal(int pidfd, int sig,
                           siginfo_t __user *info,
                           unsigned int flags);

+/* User defined syscall by Recolic Keghart <root@recolic.net>,
+ * as a naive homework.
+ **/
+asmlinkage long sys_hust_cp(const char __user *srcfname, const
char __user *dstfname, umode_t dst_mode);
+/* User defined syscall end */
+
 /*
  * Architecture-specific system calls
  */
diff --git a/include/uapi/asm-generic/unistd.h
b/include/uapi/asm-generic/unistd.h
index 1fc8faa6e..e7147a89f 100644
--- a/include/uapi/asm-generic/unistd.h
+++ b/include/uapi/asm-generic/unistd.h
@@ -850,9 +850,11 @@ __SYSCALL(__NR_pidfd_open,
sys_pidfd_open)
 #define __NR_clone3 435
 __SYSCALL(__NR_clone3, sys_clone3)
 #endif
+#define __NR_hust_cp 436
+__SYSCALL(__NR_hust_cp, sys_hust_cp)

 #undef __NR_syscalls
-#define __NR_syscalls 436
+#define __NR_syscalls 437

 /*
  * 32 bit systems traditionally used different
diff --git a/kernel/sys_ni.c b/kernel/sys_ni.c
index 3b69a560a..2e38fe870 100644
--- a/kernel/sys_ni.c
+++ b/kernel/sys_ni.c
@@ -472,3 +472,6 @@ COND_SYSCALL(setuid16);
```

```
 /* restartable sequence */
 COND_SYSCALL(rseq);
+
+/* HUST homework by recolic */
+COND_SYSCALL(hust_cp);
```

3 实验三 增加设备驱动程序

3.1 实验目的

掌握 Linux 操作系统的使用方法；

了解 Linux 系统内核代码结构；

掌握实例操作系统的实现方法。

3.2 实验内容

掌握增加设备驱动程序的方法。通过模块方法，增加一个新的字符设备驱动程序，

其功能可以简单,基于内核缓冲区。

基本要求：演示实现字符设备读、写；

选择：键盘缓冲区，不同进程、追加、读取。

3.3 实验设计

3.3.1 开发环境

```
recolic@RECOLICMPC
OS: Manjaro 20.0 Lysia
Kernel: x86_64 Linux 4.19.120-1-MANJARO
Uptime: 21m
Packages: 1852
Shell: fish 3.1.1
Resolution: 1920x1080
DE: GNOME 3.36.2
WM: Mutter
WM Theme:
GTK Theme: Adwaita [GTK2/3]
Icon Theme: Adwaita
```

Font: Cantarell 11
Disk: 65G / 111G (62%)

CPU: Intel Core m3-7Y30 @ 4x 2.6GHz [61.0°C]

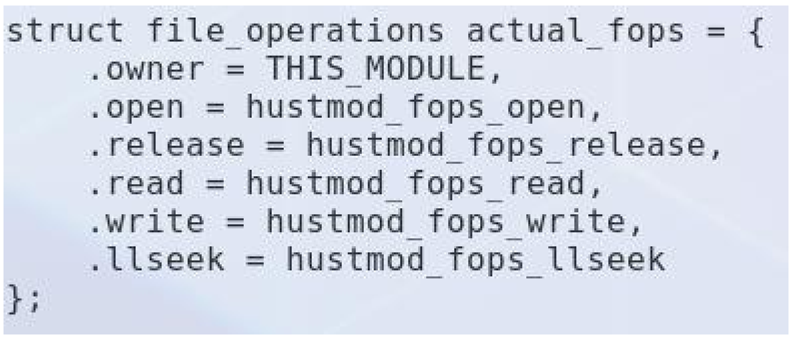
GPU: Intel Corporation HD Graphics 615 (rev 02)
RAM: 2968MiB / 3827MiB

3.3.2 实验设计

要进行字符驱动程序的开发，首先需要阅读内核开发者提供的驱动程序开发文档．文档描述, Linux 的内核模块需要提供 init 和 exit 函数．我们在这两个函数中，分别对我们的字符设备进行初始化，注册和销毁.

对于字符设备的驱动程序，我们使用 alloc_chrdev_region 来分配设备号，然后使用 cdev_init 来初始化字符设备．当一切准备就绪后，使用 cdev_add 来通知内核，这个字符设备已经可以被使用了．一旦 cdev_add 被调用，内核便可能使用任何并行的方式来访问我们的设备，因此必须在所有数据结构都初始化完成后再调用 cdev_add.

我们准备了一个 file_operations 结构，并提供了处理每一类文件操作的函数体．在我们的实现中, open 函数调用 filp_open 来打开下层的文件，然后将我们的私有数据结构初始化，并放进 private_data 中．close 函数调用 filp_close 来关闭下层的文件．随后, read, write, llseek 操作均可以直接调用 vfs 的对应函数进行实现．至此，我们的字符设备就可以像一个普通文件一样进行操作了.



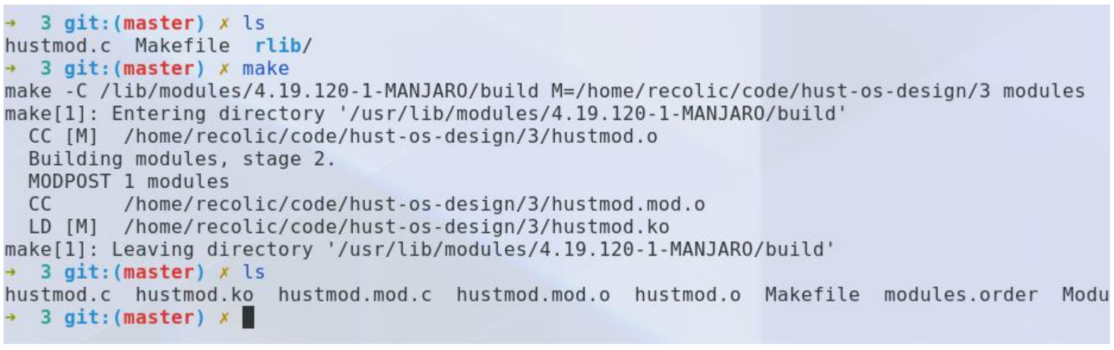
```
struct file_operations actual_fops = {
    .owner = THIS_MODULE,
    .open = hustmod_fops_open,
    .release = hustmod_fops_release,
    .read = hustmod_fops_read,
    .write = hustmod_fops_write,
    .llseek = hustmod_fops_llseek
};
```

3.4 实验调试

3.4.1 实验步骤

首先，按照内核开发者的推荐，创建一个 Makefile. 这里巧妙的利用了 ifneq ($(KERNELRELEASE),) 语句来判断自己是否是被内核目录下的 Makefile 所调用.

随后，按照实验设计思路对具体代码进行实现. 确认代码无误后运行 make 命令. 可以观察到, make 自动进入了内核提供的modules编译目录，自动正确的完成了编译过程. 此步骤的操作如下图所示.



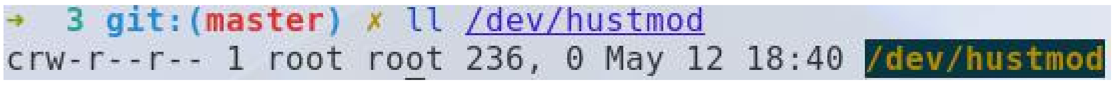
```
➜  3 git:(master) ✗ ls
hustmod.c  Makefile  rlib/
➜  3 git:(master) ✗ make
make -C /lib/modules/4.19.120-1-MANJARO/build M=/home/recolic/code/hust-os-design/3 modules
make[1]: Entering directory '/usr/lib/modules/4.19.120-1-MANJARO/build'
  CC [M]  /home/recolic/code/hust-os-design/3/hustmod.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/recolic/code/hust-os-design/3/hustmod.mod.o
  LD [M]  /home/recolic/code/hust-os-design/3/hustmod.ko
make[1]: Leaving directory '/usr/lib/modules/4.19.120-1-MANJARO/build'
➜  3 git:(master) ✗ ls
hustmod.c  hustmod.ko  hustmod.mod.c  hustmod.mod.o  hustmod.o  Makefile  modules.order  Modu
➜  3 git:(master) ✗ █
```

可以观察到, hustmod.ko 已经被编译好，在当前目录下了. 由于我们的模块没有依赖关系，所以我们可以直接运行 insmod hustmod.ko 来进行安装.

随后，我们运行 dmesg，观看内核日志，发现以下输出内容:

```
[ 2387.897724] HUSTMOD: init
[ 2387.897728] HUSTMOD: device id MAJOR:MINOR = 236:0
```

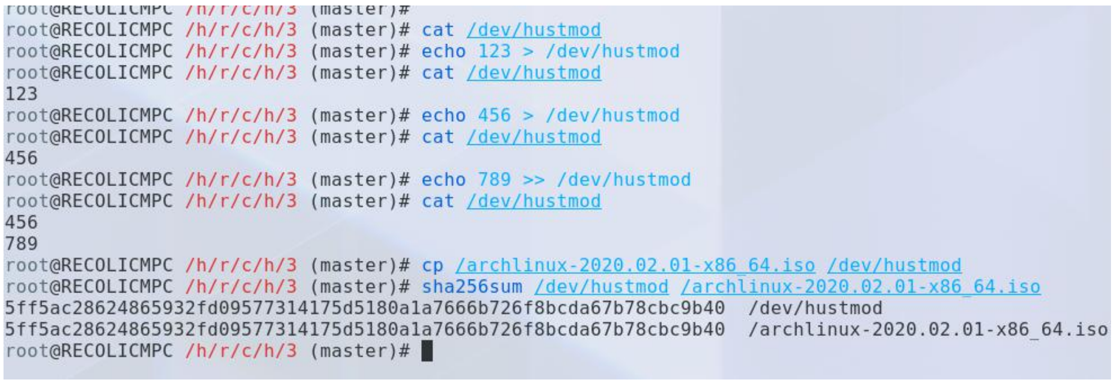
表明模块已经被成功加载. 这是我们运行 mknod /dev/hustmod c 236 0 命令, 将刚刚创建的字符设备映射到 vfs 上, 可供我们进行操作. 此时我们的字符设备已经能正常工作了, 只需对/dev/hustmod 这个文件进行操作即可.



```
→  3 git:(master) ✗ ll /dev/hustmod
crw-r--r-- 1 root root 236, 0 May 12 18:40 /dev/hustmod
```

3.4.2 实验调试及心得

在完成字符设备驱动程序的加载, 和字符设备文件的创建后, 我们对这个新的字符设备文件/dev/hustmod 进行如下一系列测试. 可以看到, 字符设备的读, 写, 追加写等操作均能正常完成, 用大文件进行的读写测试也能正常通过.



```
root@RECOLICMPC /h/r/c/h/3 (master)#
root@RECOLICMPC /h/r/c/h/3 (master)# cat /dev/hustmod
root@RECOLICMPC /h/r/c/h/3 (master)# echo 123 > /dev/hustmod
root@RECOLICMPC /h/r/c/h/3 (master)# cat /dev/hustmod
123
root@RECOLICMPC /h/r/c/h/3 (master)# echo 456 > /dev/hustmod
root@RECOLICMPC /h/r/c/h/3 (master)# cat /dev/hustmod
456
root@RECOLICMPC /h/r/c/h/3 (master)# echo 789 >> /dev/hustmod
root@RECOLICMPC /h/r/c/h/3 (master)# cat /dev/hustmod
456
789
root@RECOLICMPC /h/r/c/h/3 (master)# cp /archlinux-2020.02.01-x86_64.iso /dev/hustmod
root@RECOLICMPC /h/r/c/h/3 (master)# sha256sum /dev/hustmod /archlinux-2020.02.01-x86_64.iso
5ff5ac28624865932fd09577314175d5180a1a7666b726f8bcda67b78cbc9b40  /dev/hustmod
5ff5ac28624865932fd09577314175d5180a1a7666b726f8bcda67b78cbc9b40  /archlinux-2020.02.01-x86_64.iso
root@RECOLICMPC /h/r/c/h/3 (master)# █
```

在本次实验中, 我们像实验二一样, 依赖于内核开发者提供的完善的文档完成了实验. 内核开发者预留了非常简洁和易于开发的接口, 允许用户自由的加载和卸载模块, 极大的提升了 linux 内核的可扩展性和主代码仓库的稳定性. 这启发着我们, 在以后的软件设计中, 也要秉承模块化的良好设计理念, 对代码未来的可维护性和可扩展性做出良好的规划.

附录 实验代码

```
// hustmod.c
#include <linux/init.h>
#include <linux/module.h>
```

```
#include <linux/cdev.h>
#include <linux/syscalls.h>
#include <linux/file.h>
#include "rlib/macro.hpp"
MODULE_LICENSE("Dual BSD/GPL");

#define DATA_FNAME "/.recolic-hust.buffer"
#define DEV_COUNT 1
dev_t dev_id;
struct cdev actual_cdev;

#define          CONTEXT_PTR          ((struct          session_context
*)filep->private_data)
struct session_context {
    struct file *filep;
};

int hustmod_fops_open(struct inode *inode, struct file *filep) {
    struct file *actual_filep = filp_open(DATA_FNAME, filep->f_flags |
O_CREAT, 0000);
    if(IS_ERR(actual_filep)) {
        printk(KERN_ALERT "HUSTMOD: filp_open failed.\n");
        return -1;
    }

    filep->private_data = vmalloc(sizeof(struct session_context));
    if(filep->private_data == NULL) {
        printk(KERN_ALERT "HUSTMOD: vmalloc failed.\n");
        return -1;
    }

    CONTEXT_PTR->filep = actual_filep;
    return 0;
}

int hustmod_fops_release(struct inode *inode, struct file *filep) {
    filp_close(CONTEXT_PTR->filep, NULL);
    vfree(filep->private_data);
    return 0;
}

#define RLIB_IMPL_FALLBACK_VFS(vfs_func, ...) \
    __auto_type ret = vfs_func(CONTEXT_PTR->filep, __VA_ARGS__); \
    if(ret      <      0)      printk(KERN_ALERT      "HUSTMOD:      "
RLIB_MACRO_TO_CSTR(vfs_func) " failed.\n"); \
	return ret;

ssize_t hustmod_fops_read(struct file *filep, char __user *buf, size_t
count, loff_t *offset) {
	RLIB_IMPL_FALLBACK_VFS(vfs_read, buf, count, offset)
}

ssize_t hustmod_fops_write(struct file *filep, const char __user *buf,
size_t count, loff_t *offset) {
	RLIB_IMPL_FALLBACK_VFS(vfs_write, buf, count, offset)
}

loff_t hustmod_fops_llseek(struct file *filep, loff_t offset, int whence) {
	RLIB_IMPL_FALLBACK_VFS(vfs_llseek, offset, whence)
}

struct file_operations actual_fops = {
	.owner = THIS_MODULE,
	.open = hustmod_fops_open,
	.release = hustmod_fops_release,
	.read = hustmod_fops_read,
	.write = hustmod_fops_write,
	.llseek = hustmod_fops_llseek
};

static int hustmod_init(void) {
	printk(KERN_INFO "HUSTMOD: init\n");
	int err = 0;

	err = alloc_chrdev_region(&dev_id, 0, DEV_COUNT,
"hustmod_dev");
	if(err) {
		printk(KERN_ALERT "HUSTMOD: alloc_chrdev_region
returns %d", err);
		return err;
	}
	printk(KERN_INFO "HUSTMOD: device id MAJOR:MINOR = %d:%d",
MAJOR(dev_id), MINOR(dev_id));

	cdev_init(&actual_cdev, &actual_fops);
	actual_cdev.owner = THIS_MODULE;
	actual_cdev.ops = &actual_fops;
	err = cdev_add(&actual_cdev, dev_id, 1);
	if(err) {
		printk(KERN_ALERT "HUSTMOD: cdev_add returns %d", err);
		return err;
	}

	return err;
}

static void hustmod_exit(void) {
	printk(KERN_INFO "HUSTMOD: exit\n");
	cdev_del(&actual_cdev);
	unregister_chrdev_region(dev_id, DEV_COUNT);
}

module_init(hustmod_init);
module_exit(hustmod_exit);

//## Makefile
ifneq ($(KERNELRELEASE),)
obj-m := hustmod.o
ccflags-y := -std=gnu99 -Wno-declaration-after-statement
else
KERNELDIR ?= /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

hustmod.ko:
	$(MAKE) -C $(KERNELDIR) M=$(PWD) modules

clean:
	$(MAKE) -C $(KERNELDIR) M=$(PWD) clean

ins: hustmod.ko
	insmod hustmod.ko
	mknod /dev/hustmod0 c $$(dmesg | grep 'HUSTMOD: device id' |
tail -n 1 | sed 's/^.*MAJOR:MINOR = //g' | sed 's/:/ /')

rm:
	rmmod hustmod
	rm -f /dev/hustmod0

endif
```

4 实验四 设计并实现一个文件系统

4.1 实验目的

掌握 Linux 操作系统的使用方法；

了解 Linux 系统内核代码结构；

掌握实例操作系统的实现方法。

4.2 实验内容

(1)基于一大文件(10M 或 100M)，模拟磁盘；

(2)格式化，建立文件系统管理数据结构；

(3)基本操作，实现文件、目录相关操作。

4.3 实验设计

4.3.1 开发环境

```
recolic@RECOLICMPC
OS: Manjaro 20.0 Lysia
Kernel: x86_64 Linux 4.19.120-1-MANJARO
Uptime: 21m
Packages: 1852
Shell: fish 3.1.1
Resolution: 1920x1080
DE: GNOME 3.36.2
WM: Mutter
WM Theme:
GTK Theme: Adwaita [GTK2/3]
Icon Theme: Adwaita
Font: Cantarell 11
Disk: 65G / 111G (62%)
```

CPU: Intel Core m3-7Y30 @ 4x 2.6GHz [61.0°C]

GPU: Intel Corporation HD Graphics 615 (rev 02)
RAM: 2968MiB / 3827MiB

4.3.2 实验设计

显然, Linux 对自定义文件系统已经有完善而成熟的支持. Linux 的 vfs 支持:

1. 利用内核模块创建自定义文件系统.

2. 将文件映射为块设备，即所谓的 loop 设备.

这样我们只需要按内核开发者的文档指导，增加一个新的文件系统，然后将一个文件挂载为 loop 设备，并在这个新的块设备上格式化自己的文件系统. 我们暂时把新文件系统命名为 rfs，我们需要实现以下两个程序:

1. 一个内核模块 rfs.ko. 模块初始化时，注册文件系统. 模块卸载时，移除文件系统. 模块需要包含对文件系统内的文件进行处理的代码.

2. 一个 mkfs.rfs 程序. 它负责在一个块设备上初始化 rfs 的相关数据结构，例如超级块, inode 等结构.

根据内核开发者文档的指导, rfs.ko 需要实现以下几个大功能:

1. 总体逻辑. 负责向内核注册和移除文件系统，注册和移除模块相关逻辑. 这里含有很多函数指针，指向 super_operations，inode_operations，file_operations 所需要实现的诸多函数.

2. 超级块逻辑. 这部分函数都被 super_operations 引用，负责管理文件系统的超级块. 这包括 destroy_inode 和 put_super.

3. inode 逻辑. 这部分函数都被 inode_operations 引用，负责管理文件系统的 inode. 在 linux 4.10 以后的内核中，这包括 create, mkdir 和 lookup 操

作．这部分操作是整个文件系统中最为复杂的，我们还存在 alloc_inode, fill_inode, get/save inode, add dir/file record 等辅助函数，帮助完成 inode 的全部管理工作.

4. 文件逻辑．在 linux 4.10 以后的内核中，在 dir_operations 新增加了 iterate 这个 api. 同时，我们的 file_operations 支持最基本的 read 和 write.

4.4 实验调试

4.4.1 实验步骤

首先实现 krfs.c，在这个文件中实现主要框架逻辑. 然后将用到的函数指针，函数声明放到 krfs.h，并在 super.c, inode.c, file.c, dir.c 分别提供函数的具体实现.

然后实现 mkfs.rfs.c，在这个文件中提供一个 main 函数，分别将初始的 inode 和 super block 写入块设备(其实是文件)，并自动创建两个用于测试的初始文件，这样块设备就被格式化完成了.

然后运行 make，它将编译生成内核模块 rfs.ko 和可执行文件 mkfs.rfs. 然后依次运行以下命令:

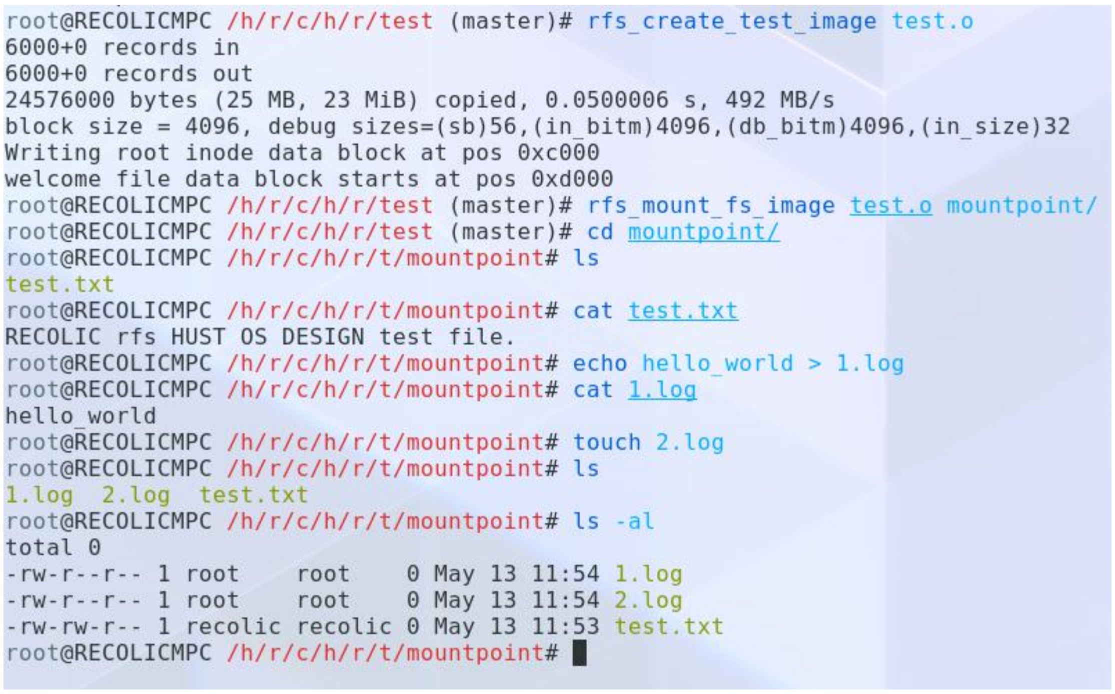
```
insmod ./rfs.ko #  插入内核模块
mount -o loop,owner,group,users -t rfs 4GB.file mountpoint/ #  挂载 loop 设备
cd mountpoint/
```

这时我们已经在新的文件系统内了.

4.4.2 实验调试及心得

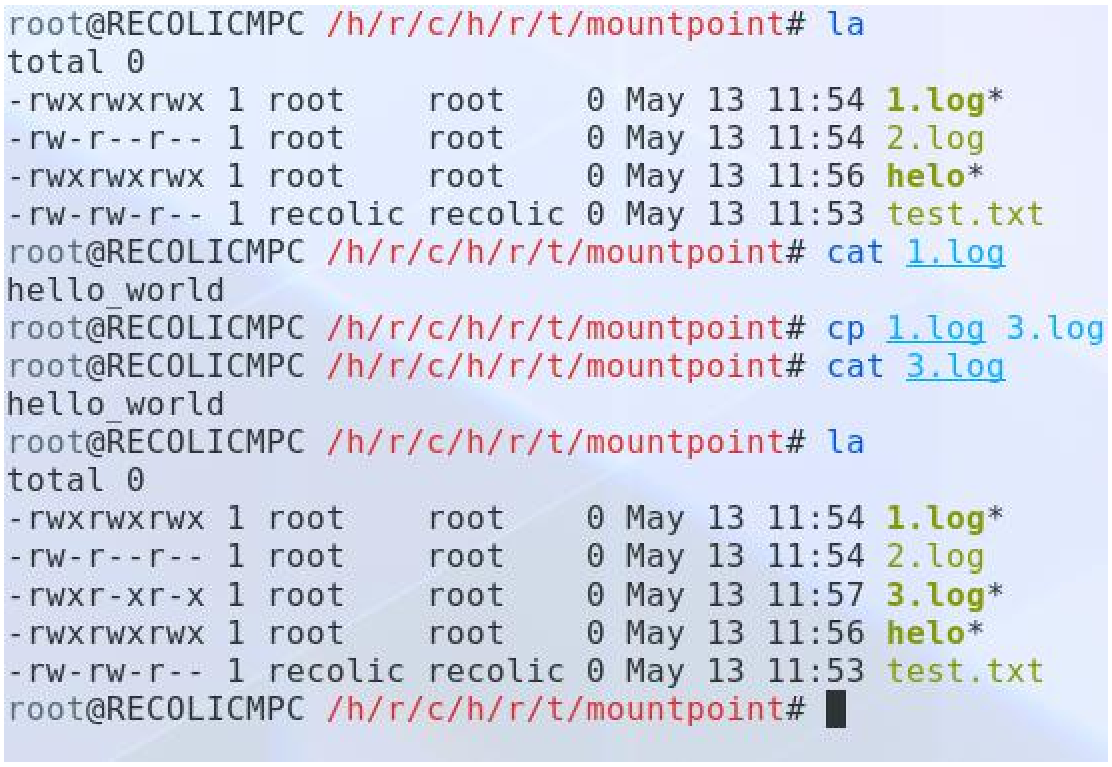
在调试过程中，为了简化操作，我实现了 bootstrap.fish 这个脚本．只需在测试时 source 这个脚本，即可使用 rfs_create_test_image, rfs_mount_fs_image 和 rfs_unmount_fs 等便捷的命令．

在新文件系统内，我们可以尝试 cat 一个文件，或者 echo 一些内容到一个文件，进行测试．测试过程如下图所示．



```
root@RECOLICMPC /h/r/c/h/r/test (master)# rfs_create_test_image test.o
6000+0 records in
6000+0 records out
24576000 bytes (25 MB, 23 MiB) copied, 0.0500006 s, 492 MB/s
block size = 4096, debug sizes=(sb)56,(in_bitm)4096,(db_bitm)4096,(in_size)32
Writing root inode data block at pos 0xc000
welcome file data block starts at pos 0xd000
root@RECOLICMPC /h/r/c/h/r/test (master)# rfs_mount_fs_image test.o mountpoint/
root@RECOLICMPC /h/r/c/h/r/test (master)# cd mountpoint/
root@RECOLICMPC /h/r/c/h/r/t/mountpoint# ls
test.txt
root@RECOLICMPC /h/r/c/h/r/t/mountpoint# cat test.txt
RECOLIC rfs HUST OS DESIGN test file.
root@RECOLICMPC /h/r/c/h/r/t/mountpoint# echo hello_world > 1.log
root@RECOLICMPC /h/r/c/h/r/t/mountpoint# cat 1.log
hello_world
root@RECOLICMPC /h/r/c/h/r/t/mountpoint# touch 2.log
root@RECOLICMPC /h/r/c/h/r/t/mountpoint# ls
1.log  2.log  test.txt
root@RECOLICMPC /h/r/c/h/r/t/mountpoint# ls -al
total 0
-rw-r--r-- 1 root    root    0 May 13 11:54 1.log
-rw-r--r-- 1 root    root    0 May 13 11:54 2.log
-rw-rw-r-- 1 recolic recolic 0 May 13 11:53 test.txt
root@RECOLICMPC /h/r/c/h/r/t/mountpoint# █
```

可以看到，创建文件，读写文件功能均正常，文件的权限位与所有者功能均正常工作．



```
root@RECOLICMPC /h/r/c/h/r/t/mountpoint# la
total 0
-rwxrwxrwx 1 root    root    0 May 13 11:54 1.log*
-rw-r--r-- 1 root    root    0 May 13 11:54 2.log
-rwxrwxrwx 1 root    root    0 May 13 11:56 helo*
-rw-rw-r-- 1 recolic recolic 0 May 13 11:53 test.txt
root@RECOLICMPC /h/r/c/h/r/t/mountpoint# cat 1.log
hello_world
root@RECOLICMPC /h/r/c/h/r/t/mountpoint# cp 1.log 3.log
root@RECOLICMPC /h/r/c/h/r/t/mountpoint# cat 3.log
hello_world
root@RECOLICMPC /h/r/c/h/r/t/mountpoint# la
total 0
-rwxrwxrwx 1 root    root    0 May 13 11:54 1.log*
-rw-r--r-- 1 root    root    0 May 13 11:54 2.log
-rwxr-xr-x 1 root    root    0 May 13 11:57 3.log*
-rwxrwxrwx 1 root    root    0 May 13 11:56 helo*
-rw-rw-r-- 1 recolic recolic 0 May 13 11:53 test.txt
root@RECOLICMPC /h/r/c/h/r/t/mountpoint# 
```

在测试中可以发现，预期功能均能够正确工作.

本次实验是一个较为大型的实验，重点在于正确的实现文件系统的每一个功能. 而且网络上与此相关的资料并不多，主要依赖 linux 内核开发者文档，和一本讲 linux 内核编程的书籍. 在按照文档进行代码的实现的同时，也感叹于 linux vfs 虚拟文件系统的接口设计之高明，和良好的可扩展性. 同时，如果没有实验的要求，这种小文件系统也可以用用户态的 fuse 进行实现，进一步降低实现新文件系统的难度.

附录 实验代码

```
//# ----- dir.c
#include "krfs.h"

#if LINUX_VERSION_CODE < KERNEL_VERSION(3, 11, 0)
#error fs api changed in linux 3.11.0. Please use a better kernel to build my code!
#endif

int rfs_iterate(struct file *filp, struct dir_context *ctx) {
    RLIB_KTRACE_FUNC(iterate);

    auto inode = filp->f_path.dentry->d_inode;
    auto sb = inode->i_sb;
    auto rfs_inode = RFS_INODE(inode);

    if (ctx->pos) {
        // TODO @Sankar: we use a hack of reading pos to figure if we have filled in data.
        printk(KERN_ALERT "iterate, pos != 0.\n");
        return 0;
    }

    printk(KERN_INFO "readdir: rfs_inode->inode_no=%llu", rfs_inode->inode_no);

    if (unlikely(!S_ISDIR(rfs_inode->mode))) {
        printk(KERN_ERR "Inode %llu of dentry %s is not a directory\n", rfs_inode->inode_no,
                filp->f_path.dentry->d_name.name);
        return -ENOTDIR;
    }

    auto bh = sb_bread(sb, rfs_inode->data_block_no);
    BUG_ON(!bh);

    auto dir_record = (struct rfs_dir_record *)bh->b_data;
    for (auto i = 0; i < rfs_inode->dir_children_count; i++) {
        dir_emit(ctx,  dir_record->filename,  RFS_FILENAME_MAXLEN,  dir_record->inode_no,
DT_UNKNOWN);
        ctx->pos += sizeof(struct rfs_dir_record);
        dir_record++;
    }
    brelse(bh);

    return 0;
}
//# ----- file.c
#include "krfs.h"

ssize_t rfs_read(struct file *filp, char __user *buf, size_t len, loff_t *ppos) {
    auto inode = filp->f_path.dentry->d_inode;
    auto sb = inode->i_sb;
    auto rfs_inode = RFS_INODE(inode);

    if (*ppos >= rfs_inode->file_size) {
        return 0;
    }

    auto bh = sb_bread(sb, rfs_inode->data_block_no);
    if (!bh) {
        printk(KERN_ERR "Failed to read data block %llu\n", rfs_inode->data_block_no);
        return 0;
    }

    auto buffer = (char *)bh->b_data + *ppos;
    auto nbytes = min((size_t)(rfs_inode->file_size - *ppos), len);

    if (copy_to_user(buf, buffer, nbytes)) {
        brelse(bh);
        printk(KERN_ERR "Error copying file content to userspace buffer\n");
        return -EFAULT;
    }

    brelse(bh);
    *ppos += nbytes;
    return nbytes;
}

/* TODO We didn't use address_space/pagecache here.
   If we hook file_operations.write = do_sync_write,
   and file_operations.aio_write = generic_file_aio_write,
   we will use write to pagecache instead. */
ssize_t rfs_write(struct file *filp, const char __user *buf, size_t len, loff_t *ppos) {
    auto inode = filp->f_path.dentry->d_inode;
    auto sb = inode->i_sb;
    auto rfs_inode = RFS_INODE(inode);

    // Recolic: compilation issue, temporary disable. TODO
    // ret = generic_write_checks(filp, ppos, &len, 0);
    // if (ret) {
    //      return ret;
    // }

    auto bh = sb_bread(sb, rfs_inode->data_block_no);
    if (!bh) {
        printk(KERN_ERR "Failed to read data block %llu\n", rfs_inode->data_block_no);
        return 0;
    }

    auto buffer = (char *)bh->b_data + *ppos;
    if (copy_from_user(buffer, buf, len)) {
        brelse(bh);
        printk(KERN_ERR "Error copying file content from userspace buffer "
                        "to kernel space\n");
        return -EFAULT;
    }
    *ppos += len;

    mark_buffer_dirty(bh);
    sync_dirty_buffer(bh);
    brelse(bh);

    rfs_inode->file_size = max((size_t)(rfs_inode->file_size), (size_t)(*ppos));
    rfs_save_rfs_inode(sb, rfs_inode);

    /* TODO We didn't update file size here. To be frank I don't know how. */

    return len;
}
//# ----- inode.c
#include "krfs.h"

void rfs_destroy_inode(struct inode *inode) {
    auto rfs_inode = RFS_INODE(inode);
    printk(KERN_INFO "destroy_inode free private data of %p (%lu)\n", rfs_inode,
inode->i_ino);
    kmem_cache_free(rfs_inode_cache, rfs_inode);
}

void rfs_fill_inode(struct super_block *sb, struct inode *inode, struct rfs_inode *rfs_inode) {
    inode->i_mode = rfs_inode->mode;
    inode->i_sb = sb;
    inode->i_ino = rfs_inode->inode_no;
    inode->i_op = &rfs_inode_ops;
    // TODO hope we can use rfs_inode to store timespec
    inode->i_atime = inode->i_mtime = inode->i_ctime = current_time(inode);
    inode->i_private = rfs_inode;

    if (S_ISDIR(rfs_inode->mode)) {
        inode->i_fop = &rfs_dir_operations;
    } else if (S_ISREG(rfs_inode->mode)) {
        inode->i_fop = &rfs_file_operations;
    } else {
        printk(KERN_WARNING "Inode %lu is neither a directory nor a regular file",
inode->i_ino);
        inode->i_fop = NULL;
    }

    /* TODO rfs_inode->file_size seems not reflected in inode */
}

/* TODO I didn't implement any function to dealloc rfs_inode */
int rfs_alloc_rfs_inode(struct super_block *sb, uint64_t *out_inode_no) {
    int ret = -ENOSPC;
```

```
	mutex_lock(&rfs_sb_lock);

	auto bh = sb_bread(sb, RFS_INODE_BITMAP_BLOCK_NO);
	BUG_ON(!bh);
	auto rfs_sb = RFS_SB(sb);

	auto bitmap = bh->b_data;
	for (auto i = 0; i < rfs_sb->inode_table_size; i++) {
		auto slot = bitmap + i / BITS_IN_BYTE;
		auto needle = 1 << (i % BITS_IN_BYTE);
		if (0 == (*slot & needle)) {
			*out_inode_no = i;
			*slot |= needle;
			rfs_sb->inode_count += 1;
			ret = 0;
			break;
		}
	}
	// Booms if inode buffer is full.
	// locking critical section is too large, but Im too lazy
	//	to have it optimized.

	mark_buffer_dirty(bh);
	sync_dirty_buffer(bh);
	brelse(bh);
	rfs_save_sb(sb);

	mutex_unlock(&rfs_sb_lock);
	return ret;
}

struct rfs_inode *rfs_get_rfs_inode(struct super_block *sb, uint64_t inode_no) {
	struct buffer_head *bh;
	struct rfs_inode *inode;
	struct rfs_inode *inode_buf;

	bh = sb_bread(sb, RFS_INODE_TABLE_START_BLOCK_NO + 
RFS_INODE_BLOCK_OFFSET(sb, inode_no));
	BUG_ON(!bh);

	inode = (struct rfs_inode *)(bh->b_data + RFS_INODE_BYTE_OFFSET(sb, inode_no));
	inode_buf = kmem_cache_alloc(rfs_inode_cache, GFP_KERNEL);
	memcpy(inode_buf, inode, sizeof(*inode_buf));

	brelse(bh);
	return inode_buf;
}

void rfs_save_rfs_inode(struct super_block *sb, struct rfs_inode *inode_buf) {
	auto inode_no = inode_buf->inode_no;
	auto bh = sb_bread(sb, RFS_INODE_TABLE_START_BLOCK_NO + 
RFS_INODE_BLOCK_OFFSET(sb, inode_no));
	BUG_ON(!bh);

	auto inode = (struct rfs_inode *)(bh->b_data + RFS_INODE_BYTE_OFFSET(sb, inode_no));
	memcpy(inode, inode_buf, sizeof(*inode));

	mark_buffer_dirty(bh);
	sync_dirty_buffer(bh);
	brelse(bh);
}

int rfs_add_dir_record(struct super_block *sb, struct inode *dir, struct dentry *dentry,
			struct inode *inode) {
	auto parent_rfs_inode = RFS_INODE(dir);
	if (unlikely(parent_rfs_inode->dir_children_count >= RFS_DIR_MAX_RECORD(sb))) {
		return -ENOSPC;
	}
```

```
    auto bh = sb_bread(sb, parent_rfs_inode->data_block_no);
    BUG_ON(!bh);

    auto dir_record = (struct rfs_dir_record *)bh->b_data;
    dir_record += parent_rfs_inode->dir_children_count;
    dir_record->inode_no = inode->i_ino;
    strcpy(dir_record->filename, dentry->d_name.name);

    mark_buffer_dirty(bh);
    sync_dirty_buffer(bh);
    brelse(bh);

    parent_rfs_inode->dir_children_count += 1;
    rfs_save_rfs_inode(sb, parent_rfs_inode);

    return 0;
}

int rfs_alloc_data_block(struct super_block *sb, uint64_t *out_data_block_no) {
    int ret = -ENOSPC;

    auto rfs_sb = RFS_SB(sb);
    mutex_lock(&rfs_sb_lock);

    auto bh = sb_bread(sb, RFS_DATA_BLOCK_BITMAP_BLOCK_NO);
    BUG_ON(!bh);

    auto bitmap = bh->b_data;
    for (auto i = 0; i < rfs_sb->data_block_table_size; i++) {
        auto slot = bitmap + i / BITS_IN_BYTE;
        auto needle = 1 << (i % BITS_IN_BYTE);
        if (0 == (*slot & needle)) {
            *out_data_block_no = RFS_DATA_BLOCK_TABLE_START_BLOCK_NO(sb) + i;
            *slot |= needle;
            rfs_sb->data_block_count += 1;
            ret = 0;
            break;
        }
    }

    mark_buffer_dirty(bh);
    sync_dirty_buffer(bh);
    brelse(bh);
    rfs_save_sb(sb);

    mutex_unlock(&rfs_sb_lock);
    return ret;
}

int rfs_create_inode(struct inode *dir, struct dentry *dentry, umode_t mode) {
    auto sb = dir->i_sb;
    auto rfs_sb = RFS_SB(sb);

    /* Create rfs_inode */
    uint64_t inode_no;
    auto ret = rfs_alloc_rfs_inode(sb, &inode_no);
    if (0 != ret) {
        printk(KERN_ERR "Unable to allocate on-disk inode. "
                        "Is inode table full? "
                        "Inode count: %llu\n",
               rfs_sb->inode_count);
        return -ENOSPC;
    }
    auto rfs_inode = (struct rfs_inode *)kmem_cache_alloc(rfs_inode_cache, GFP_KERNEL);
    rfs_inode->inode_no = inode_no;
    rfs_inode->mode = mode;
    if (S_ISDIR(mode)) {
        rfs_inode->dir_children_count = 0;
    } else if (S_ISREG(mode)) {
        rfs_inode->file_size = 0;
    } else {
        printk(KERN_WARNING "Inode %llu is neither a directory nor a regular file",
inode_no);
    }

    /* Allocate data block for the new rfs_inode */
    ret = rfs_alloc_data_block(sb, &rfs_inode->data_block_no);
    if (0 != ret) {
        printk(KERN_ERR "Unable to allocate on-disk data block. "
                                "Is data block table full? "
                                "Data block count: %llu\n",
                rfs_sb->data_block_count);
        return -ENOSPC;
    }

    /* Create VFS inode */
    auto inode = new_inode(sb);
    if (!inode) {
        return -ENOMEM;
    }
    rfs_fill_inode(sb, inode, rfs_inode);

    /* Add new inode to parent dir */
    ret = rfs_add_dir_record(sb, dir, dentry, inode);
    if (0 != ret) {
        printk(KERN_ERR "Failed to add inode %lu to parent dir %lu\n", inode->i_ino,
dir->i_ino);
        return -ENOSPC;
    }

    inode_init_owner(inode, dir, mode);
    d_add(dentry, inode);

    /* TODO we should free newly allocated inodes when error occurs */

    return 0;
}

int rfs_create(struct inode *dir, struct dentry *dentry, umode_t mode, bool excl) {
    RLIB_KTRACE_FUNC(rfs_create);
    return rfs_create_inode(dir, dentry, mode);
}

int rfs_mkdir(struct inode *dir, struct dentry *dentry, umode_t mode) {
    RLIB_KTRACE_FUNC(rfs_mkdir);
    /* @Sankar: The mkdir callback does not have S_IFDIR set.
       Even ext2 sets it explicitly. Perhaps this is a bug */
    mode |= S_IFDIR;
    return rfs_create_inode(dir, dentry, mode);
}

struct dentry *rfs_lookup(struct inode *dir, struct dentry *child_dentry, unsigned int flags) {
    RLIB_KTRACE_FUNC(rfs_lookup);

    auto parent_rfs_inode = RFS_INODE(dir);
    auto sb = dir->i_sb;
    auto bh = sb_bread(sb, parent_rfs_inode->data_block_no);
    BUG_ON(!bh);

    auto dir_record = (struct rfs_dir_record *)bh->b_data;

    for (auto i = 0; i < parent_rfs_inode->dir_children_count; i++) {
        printk(KERN_INFO "rfs_lookup: i=%d, dir_record->filename=%s,
child_dentry->d_name.name=%s",
                i, dir_record->filename, child_dentry->d_name.name); // TODO
        if (0 == strcmp(dir_record->filename, child_dentry->d_name.name)) {
            auto rfs_child_inode = rfs_get_rfs_inode(sb, dir_record->inode_no);
```

```
			auto child_inode = new_inode(sb);
			if (!child_inode) {
				printk(KERN_ERR "Cannot create new inode. No memory.\n");
				return NULL;
			}
			rfs_fill_inode(sb, child_inode, rfs_child_inode);
			inode_init_owner(child_inode, dir, rfs_child_inode->mode);
			d_add(child_dentry, child_inode);
			return NULL;
		}
		dir_record++;
	}

	printk(KERN_ERR	"No	inode	found	for	the	filename:	%s\n",
child_dentry->d_name.name);
	return NULL;
}
//# ----- krfs.c
#include "krfs.h"

DEFINE_MUTEX(rfs_sb_lock);

struct file_system_type rfs_fs_type = {
	.owner = THIS_MODULE,
	.name = "rfs",
	.mount = rfs_mount,
	.kill_sb = rfs_kill_superblock,
	.fs_flags = FS_REQUIRES_DEV,
};

const struct super_operations rfs_sb_ops = {
	.destroy_inode = rfs_destroy_inode,
	.put_super = rfs_put_super,
};

const struct inode_operations rfs_inode_ops = {
	.create = rfs_create,
	.mkdir = rfs_mkdir,
	.lookup = rfs_lookup,
};

const struct file_operations rfs_dir_operations = {
	.owner = THIS_MODULE,
	.iterate = rfs_iterate,
};

const struct file_operations rfs_file_operations = {
	.read = rfs_read,
	.write = rfs_write,
};

struct kmem_cache *rfs_inode_cache = NULL;

static int __init rfs_init(void) {
	rfs_inode_cache = kmem_cache_create("rfs_inode_cache", sizeof(struct rfs_inode), 0,
						(SLAB_RECLAIM_ACCOUNT	|	SLAB_MEM_SPREAD),
NULL);
	if (!rfs_inode_cache) {
		return -ENOMEM;
	}

	int ret = register_filesystem(&rfs_fs_type);

	if (likely(0 == ret)) {
		printk(KERN_INFO "Sucessfully registered rfs\n");
	} else {
		printk(KERN_ERR "Failed to register rfs. Error code: %d\n", ret);
	}
```

```
    return ret;
}

static void __exit rfs_exit(void) {
    kmem_cache_destroy(rfs_inode_cache);

    int ret = unregister_filesystem(&rfs_fs_type);

    if (likely(0 == ret)) {
        printk(KERN_INFO "Sucessfully unregistered rfs\n");
    } else {
        printk(KERN_ERR "Failed to unregister rfs. Error code: %d\n", ret);
    }
}

module_init(rfs_init);
module_exit(rfs_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("accelazh");
//# ----- krfs.h
#ifndef __KRFS_H__
#define __KRFS_H__

/* krfs.h defines symbols to work in kernel space */

#include <linux/blkdev.h>
#include <linux/buffer_head.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/namei.h>
#include <linux/parser.h>
#include <linux/random.h>
#include <linux/slab.h>
#include <linux/time.h>
#include <linux/version.h>

#include "rfs.h"

/* Declare operations to be hooked to VFS */

extern struct file_system_type rfs_fs_type;
extern const struct super_operations rfs_sb_ops;
extern const struct inode_operations rfs_inode_ops;
extern const struct file_operations rfs_dir_operations;
extern const struct file_operations rfs_file_operations;

struct dentry *rfs_mount(struct file_system_type *fs_type, int flags, const char *dev_name,
                         void *data);
void rfs_kill_superblock(struct super_block *sb);

void rfs_destroy_inode(struct inode *inode);
void rfs_put_super(struct super_block *sb);

int rfs_create(struct inode *dir, struct dentry *dentry, umode_t mode, bool excl);
struct dentry *rfs_lookup(struct inode *parent_inode, struct dentry *child_dentry,
                          unsigned int flags);
int rfs_mkdir(struct inode *dir, struct dentry *dentry, umode_t mode);

#if LINUX_VERSION_CODE < KERNEL_VERSION(3, 11, 0)
#error fs api changed in linux 3.11.0. Please use a better kernel to build my code!
#endif
int rfs_iterate(struct file *filp, struct dir_context *ctx);

ssize_t rfs_read(struct file *filp, char __user *buf, size_t len, loff_t *ppos);
ssize_t rfs_write(struct file *filp, const char __user *buf, size_t len, loff_t *ppos);

extern struct kmem_cache *rfs_inode_cache;
```

```
/* Helper functions */

// To translate VFS superblock to rfs superblock
static inline struct rfs_superblock *RFS_SB(struct super_block *sb) { return sb->s_fs_info; }
static inline struct rfs_inode *RFS_INODE(struct inode *inode) { return inode->i_private; }

static inline uint64_t RFS_INODES_PER_BLOCK(struct super_block *sb) {
	struct rfs_superblock *rfs_sb;
	rfs_sb = RFS_SB(sb);
	return RFS_INODES_PER_BLOCK_HSB(rfs_sb);
}

// Given the inode_no, calcuate which block in inode table contains the corresponding inode
static inline uint64_t RFS_INODE_BLOCK_OFFSET(struct super_block *sb, uint64_t inode_no)
{
	return inode_no / RFS_INODES_PER_BLOCK_HSB(RFS_SB(sb));
}
static inline uint64_t RFS_INODE_BYTE_OFFSET(struct super_block *sb, uint64_t inode_no) {
	return (inode_no % RFS_INODES_PER_BLOCK_HSB(RFS_SB(sb))) * sizeof(struct
rfs_inode);
}

static inline uint64_t RFS_DIR_MAX_RECORD(struct super_block *sb) {
	return RFS_SB(sb)->blocksize / sizeof(struct rfs_dir_record);
}

// From which block does data blocks start
static inline uint64_t RFS_DATA_BLOCK_TABLE_START_BLOCK_NO(struct super_block *sb) {
	return RFS_DATA_BLOCK_TABLE_START_BLOCK_NO_HSB(RFS_SB(sb));
}

void rfs_save_sb(struct super_block *sb);

// functions to operate inode
void rfs_fill_inode(struct super_block *sb, struct inode *inode, struct rfs_inode *rfs_inode);
int rfs_alloc_rfs_inode(struct super_block *sb, uint64_t *out_inode_no);
struct rfs_inode *rfs_get_rfs_inode(struct super_block *sb, uint64_t inode_no);
void rfs_save_rfs_inode(struct super_block *sb, struct rfs_inode *inode);
int rfs_add_dir_record(struct super_block *sb, struct inode *dir, struct dentry *dentry,
                       struct inode *inode);
int rfs_alloc_data_block(struct super_block *sb, uint64_t *out_data_block_no);
int rfs_create_inode(struct inode *dir, struct dentry *dentry, umode_t mode);

#endif /*__KRFS_H__*/
//# ----- Makefile
obj-m := rfs.o
rfs-objs := krfs.o super.o inode.o dir.o file.o
ccflags-y := -std=gnu99 -Wno-declaration-after-statement

CFLAGS_krfs.o := -DDEBUG
CFLAGS_super.o := -DDEBUG
CFLAGS_inode.o := -DDEBUG
CFLAGS_dir.o := -DDEBUG
CFLAGS_file.o := -DDEBUG

all: ko mkfs.rfs

ko:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

mkfs.rfs_SOURCES:
	mkfs.rfs.c rfs.h

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
	rm mkfs.rfs
//# ----- mkfs.rfs.c
#include <assert.h>
```

```
#include <fcntl.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

#include "rfs.h"

int main(int argc, char *argv[]) {
    auto fd = open(argv[1], O_RDWR);
    if (fd == -1) {
        perror("Error opening the device");
        return -1;
    }

    // construct superblock
    struct rfs_superblock rfs_sb = {
        .version = 1,
        .magic = RFS_MAGIC,
        .blocksize = RFS_DEFAULT_BLOCKSIZE,
        .inode_table_size = RFS_DEFAULT_INODE_TABLE_SIZE,
        .inode_count = 2,
        .data_block_table_size = RFS_DEFAULT_DATA_BLOCK_TABLE_SIZE,
        .data_block_count = 2,
    };

    // construct inode bitmap
    char inode_bitmap[rfs_sb.blocksize];
    memset(inode_bitmap, 0, sizeof(inode_bitmap));
    inode_bitmap[0] = 1;

    // construct data block bitmap
    char data_block_bitmap[rfs_sb.blocksize];
    memset(data_block_bitmap, 0, sizeof(data_block_bitmap));
    data_block_bitmap[0] = 1;

    // construct root inode
    struct rfs_inode root_rfs_inode = {
        .mode = S_IFDIR | S_IRWXU | S_IRWXG | S_IROTH | S_IXOTH,
        .inode_no = RFS_ROOTDIR_INODE_NO,
        .data_block_no =
            RFS_DATA_BLOCK_TABLE_START_BLOCK_NO_HSB(&rfs_sb)                +
RFS_ROOTDIR_DATA_BLOCK_NO_OFFSET,
        .dir_children_count = 1,
    };

    // construct welcome file inode
    char welcome_body[] = "RECOLIC rfs HUST OS DESIGN test file.\n";
    auto welcome_inode_no = RFS_ROOTDIR_INODE_NO + 1;
    auto welcome_data_block_no_offset = RFS_ROOTDIR_DATA_BLOCK_NO_OFFSET + 1;
    struct rfs_inode welcome_rfs_inode = {
        .mode = S_IFREG | S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH,
        .inode_no = welcome_inode_no,
        .data_block_no =
            RFS_DATA_BLOCK_TABLE_START_BLOCK_NO_HSB(&rfs_sb)                +
welcome_data_block_no_offset,
        .file_size = sizeof(welcome_body),
    };

    // construct root inode data block
    struct rfs_dir_record root_dir_records[] = {
        {
            .filename = "test.txt",
            .inode_no = welcome_inode_no,
        },
    };
    auto ret = -1;
    assert(sizeof(rfs_sb) <= rfs_sb.blocksize);
    // write super block
    if (sizeof(rfs_sb) != write(fd, &rfs_sb, sizeof(rfs_sb))) {
        goto err;
    }
    if ((off_t)-1 == lseek(fd, rfs_sb.blocksize, SEEK_SET)) {
        goto err;
    }

    // write inode bitmap
    if (sizeof(inode_bitmap) != write(fd, inode_bitmap, sizeof(inode_bitmap))) {
        goto err;
    }

    // write data block bitmap
    if (sizeof(data_block_bitmap) != write(fd, data_block_bitmap, sizeof(data_block_bitmap)))
{
        goto err;
    }

    // write root inode
    if (sizeof(root_rfs_inode) != write(fd, &root_rfs_inode, sizeof(root_rfs_inode))) {
        goto err;
    }

    // write welcome file inode
    if        (sizeof(welcome_rfs_inode)        !=        write(fd,        &welcome_rfs_inode,
sizeof(welcome_rfs_inode))) {
        goto err;
    }

    printf("block size = %d, debug sizes=(sb)%d,(in_bitm)%d,(db_bitm)%d,(in_size)%d\n",
           rfs_sb.blocksize, sizeof(rfs_sb), sizeof(inode_bitmap), sizeof(data_block_bitmap),
           sizeof(root_rfs_inode));
    printf("Writing root inode data block at pos 0x%x\n",
           RFS_DATA_BLOCK_TABLE_START_BLOCK_NO_HSB(&rfs_sb) * rfs_sb.blocksize);
    printf("welcome file data block starts at pos 0x%x\n",
           (RFS_DATA_BLOCK_TABLE_START_BLOCK_NO_HSB(&rfs_sb)       +       1)       *
rfs_sb.blocksize);
    // write root inode data block
    if ((off_t)-1 ==
        lseek(fd,          RFS_DATA_BLOCK_TABLE_START_BLOCK_NO_HSB(&rfs_sb)          *
rfs_sb.blocksize, SEEK_SET)) {
        goto err;
    }
    if (sizeof(root_dir_records) != write(fd, root_dir_records, sizeof(root_dir_records))) {
        goto err;
    }

    // write welcome file inode data block
    if ((off_t)-1 ==
        lseek(fd,   (RFS_DATA_BLOCK_TABLE_START_BLOCK_NO_HSB(&rfs_sb)   +   1)   *
rfs_sb.blocksize,
              SEEK_SET)) {
        goto err;
    }
    if (sizeof(welcome_body) != write(fd, welcome_body, sizeof(welcome_body))) {
        goto err;
    }

    ret = 0; // success
err:
    close(fd);
    return ret;
}
//# ----- rfs.h
#ifndef __RFS_H__
```

```
#define __RFS_H__

#define BITS_IN_BYTE 8
#define RFS_MAGIC 0x20160105
#define RFS_DEFAULT_BLOCKSIZE 4096
#define RFS_DEFAULT_INODE_TABLE_SIZE 1024
#define RFS_DEFAULT_DATA_BLOCK_TABLE_SIZE 1024
#define RFS_FILENAME_MAXLEN 255

#include "rlib/macro.hpp"
#include "rlib/sys/os.hpp"

#if RLIB_CXX_STD > 0
#error Not supporting C++ yet.
#else
#define auto __auto_type
#endif

/* Define filesystem structures */

struct rfs_dir_record {
	char filename[RFS_FILENAME_MAXLEN];
	uint64_t inode_no;
};

struct rfs_superblock {
	uint64_t version;
	uint64_t magic;
	uint64_t blocksize;

	uint64_t inode_table_size;
	uint64_t inode_count;

	uint64_t data_block_table_size;
	uint64_t data_block_count;
};
extern struct mutex rfs_sb_lock;

struct rfs_inode {
	mode_t mode;
	uint64_t inode_no;
	uint64_t data_block_no;

	// TODO struct timespec is defined kenrel space,
	// but mkfs-rfs.c is compiled in user space
	/*struct timespec atime;
	struct timespec mtime;
	struct timespec ctime;*/

	union {
		uint64_t file_size;
		uint64_t dir_children_count;
	};
};

static const uint64_t RFS_SUPERBLOCK_BLOCK_NO = 0;
static const uint64_t RFS_INODE_BITMAP_BLOCK_NO = 1;
static const uint64_t RFS_DATA_BLOCK_BITMAP_BLOCK_NO = 2;
static const uint64_t RFS_INODE_TABLE_START_BLOCK_NO = 3;

static const uint64_t RFS_ROOTDIR_INODE_NO = 0;
// data block no is the absolute block number from start of device
// data block no offset is the relative block offset from start of data block table
static const uint64_t RFS_ROOTDIR_DATA_BLOCK_NO_OFFSET = 0;

/* Helper functions */

static inline uint64_t RFS_INODES_PER_BLOCK_HSB(struct rfs_superblock *rfs_sb) {
	return rfs_sb->blocksize / sizeof(struct rfs_inode);
```

```
}

static inline uint64_t RFS_DATA_BLOCK_TABLE_START_BLOCK_NO_HSB(struct rfs_superblock
*rfs_sb) {
    return RFS_INODE_TABLE_START_BLOCK_NO +
           rfs_sb->inode_table_size / RFS_INODES_PER_BLOCK_HSB(rfs_sb) + 1;
}

/* Debug function */
#define                                                    RLIB_KTRACE_FUNC(name)
\
    printk(KERN_ALERT "Recolic ktrace: [" RLIB_MACRO_TO_CSTR(name) "] called.\n")

#endif /*__RFS_H__*/
//# ----- rlib
cat: rlib: Is a directory
//# ----- super.c
#include "krfs.h"

static int rfs_fill_super(struct super_block *sb, void *data, int silent) {
    int ret = 0;

    auto bh = sb_bread(sb, RFS_SUPERBLOCK_BLOCK_NO);
    BUG_ON(!bh);
    auto rfs_sb = (struct rfs_superblock *)bh->b_data;
    if (unlikely(rfs_sb->magic != RFS_MAGIC)) {
        printk(KERN_ERR    "Mount    rfs    filesystem:    Wrong    magic    number    in
superblock: %llu != %llu\n",
               rfs_sb->magic, (uint64_t)RFS_MAGIC);
        goto release;
    }
    if (unlikely(sb->s_blocksize != rfs_sb->blocksize)) {
        printk(KERN_ERR "rfs seem to be formatted with mismatching blocksize: %lu\n",
               sb->s_blocksize);
        goto release;
    }

    sb->s_magic = rfs_sb->magic;
    sb->s_fs_info = rfs_sb;
    sb->s_maxbytes = rfs_sb->blocksize;
    sb->s_op = &rfs_sb_ops;

    auto root_rfs_inode = rfs_get_rfs_inode(sb, RFS_ROOTDIR_INODE_NO);
    auto root_inode = new_inode(sb);
    if (!root_inode || !root_rfs_inode) {
        ret = -ENOMEM;
        goto release;
    }
    rfs_fill_inode(sb, root_inode, root_rfs_inode);
    inode_init_owner(root_inode, NULL, root_inode->i_mode);

    sb->s_root = d_make_root(root_inode);
    if (!sb->s_root) {
        ret = -ENOMEM;
        goto release;
    }

release:
    brelse(bh);
    return ret;
}

struct dentry *rfs_mount(struct file_system_type *fs_type, int flags, const char *dev_name,
                         void *data) {
    auto ret = mount_bdev(fs_type, flags, dev_name, data, rfs_fill_super);

    if (unlikely(IS_ERR(ret))) {
        printk(KERN_ERR "Error mounting rfs.\n");
    } else {
        printk(KERN_INFO "rfs is succesfully mounted on: %s\n", dev_name);
    }

    return ret;
}

void rfs_kill_superblock(struct super_block *sb) {
    printk(KERN_INFO "rfs superblock is destroyed. Unmount succesful.\n");
    kill_block_super(sb);
}

void rfs_put_super(struct super_block *sb) { return; }

void rfs_save_sb(struct super_block *sb) {

    auto bh = sb_bread(sb, RFS_SUPERBLOCK_BLOCK_NO);
    BUG_ON(!bh);
    auto rfs_sb = RFS_SB(sb);

    bh->b_data = (char *)rfs_sb;
    mark_buffer_dirty(bh);
    sync_dirty_buffer(bh);
    brelse(bh);
}
//# ----- bootstrap.fish
#!/usr/bin/fish
## source this script, and enjoy your job.

function rfs_create_test_image
    dd bs=4096 count=6000 if=/dev/zero of=$argv[1]
    ../mkfs.rfs $argv[1]
end

function rfs_mount_fs_image
    mkdir $argv[2]
    sudo insmod ../rfs.ko
    sudo mount -o loop,owner,group,users -t rfs $argv[1] $argv[2]
end

function rfs_unmount_fs
    sudo umount $argv[1]
    sudo rmmod ../rfs.ko
end
```