

华中科技大学

操作系统原理实验报告

学 生 姓 名：	胡思勖
学 号：	U201514898
专 业：	计算机科学与技术
班 级：	计卓 1501
指 导 教 师：	张杰

2017 年 12 月 25 日

目录

实验一 进程控制实验	1
1.1 实验目的	2
1.2 实验内容	2
1.3 实验设计	2
1.4 实验过程	5
实验二 线程同步与通信实验	6
2.1 实验目的	6
2.2 实验内容	6
2.3 实验设计	6
2.4 实验过程	9
实验三 共享内存与进程同步	9
3.1 实验目的	10
3.2 实验内容	10
3.3 实验设计	10
3.4 实验过程	12
实验四 Linux 文件目录	13
4.1 实验目的	13
4.2 实验内容	13
4.3 实验设计	14
4.4 实验过程	15
附录 A 实验一代码	16
附录 B 实验二代码	19
附录 C 实验三代码	22
附录 D 实验四代码	27

实验一 进程控制实验

1.1 实验目的

1. 加深对进程的理解, 进一步认识并发执行的实质;
2. 分析进程争用资源现象, 学习解决进程互斥的方法;
3. 掌握 Linux 进程基本控制;
4. 掌握 Linux 系统中的软中断和管道通信。

1.2 实验内容

编写程序, 演示多进程并发执行和进程软中断、管道通信。

- 父进程使用系统调用 `pipe()` 建立一个管道, 然后使用系统调用 `fork()` 创建两个子进程, 子进程 1 和子进程 2;
- 子进程 1 每隔 1 秒通过管道向子进程 2 发送数据: I send you x times. (x 初值为 1, 每次发送后做加一操作), 子进程 2 从管道读出信息, 并显示在屏幕上。
- 父进程用系统调用 `signal()` 捕捉来自键盘的中断信号 (即按 `Ctrl+C` 键); 当捕捉到中断信号后, 父进程用系统调用 `Kill()` 向两个子进程发出信号, 子进程捕捉到信号后分别输出下列信息后终止:
Child Process 1 is Killed by Parent!
Child Process 2 is Killed by Parent!
- 父进程等待两个子进程终止后, 释放管道并输出如下的信息后终止
Parent Process is Killed!

1.3 实验设计

1.3.1 总体设计

实验通过三个模块实现: 父进程模块, 子进程模块 1 和子进程模块 2, 他们之间的关系如图1.1所示。父进程负责启动两个子进程, 捕捉来自终端的 `SIGINT` 信号并向子进程发送 `SIGUSR1` 信号, 子进程负责处理来自父进程的 `SIGUSR1` 信号。此外, 子进程 1 每隔一秒向子进程 2 通过管道发送消息 “I send you x times”, 子进程 2 接收到消息后则将其显示在屏幕上。

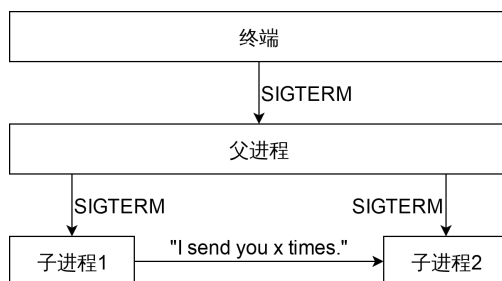


图 1.1: 实验一的三个模块及其关系

1.3.2 详细设计

对于父进程而言，其流程图如图1.2所示。在开始首先设置 signal 处理函数，使用 handlerMain 函数来处理 signal，然后创建管道，之后尝试 fork 子进程 1 和子进程 2。在 fork 的过程中如果出现错误则直接退出程序。在 fork 成功后关闭父进程的管道（父进程不需要使用管道通信），然后陷入死循环等待用户发送 SIGINT 信号结束程序。

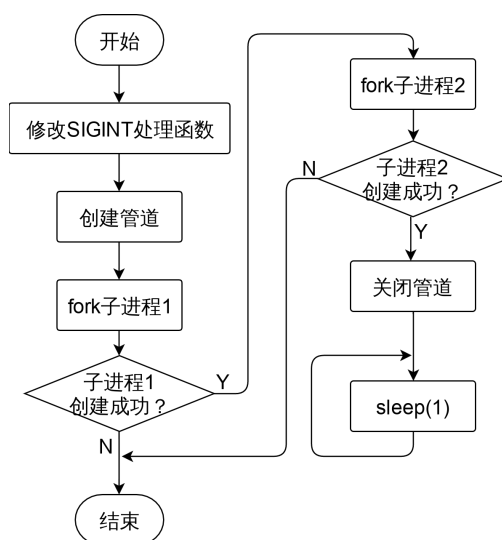


图 1.2: 主函数流程

子进程 1 的流程图如1.3所示。在启动后首先设置 SIGINT 和 SIGUSR1 的处理函数，将 SIGINT 的处理函数设置为 SIG_IGN，忽略掉来自于终端的 SIGINT 信号（由于在终端中按下 Ctrl-C 会向前台进程组中的所有进程发送 SIGINT 信号，而父进程和子进程又默认属于同一个进程组，因此需要子进程忽略掉来自终端的 SIGINT），然后将 SIGUSR1 的处理函数设置为 handlerChild1。之后直接进入死循环，每一次循环首先使用 sleep 函数等待 1 秒，然后使用 write 函数向管道中写入 “I send you x times”。

子进程 2 的流程如图1.4所示。类似的，子进程 2 在启动后使用与子进程 1 相同的处理方式处理来自终端以及来自父进程的信号，只不过其用于处理 SIGUSR1 的函数为 handlerChild2。然后陷入死循环，在每一次循环中尝试从管道中读取数据，如果读取的数据长度小于 0 说明读取出错，此时令此程序返回，否则打印读取到的数据到标准输出。

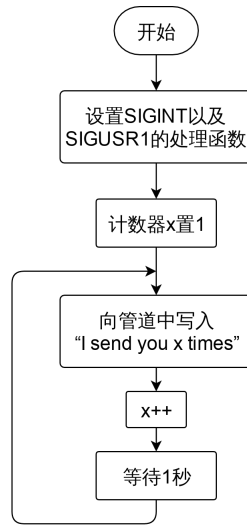


图 1.3: 子进程 1 流程

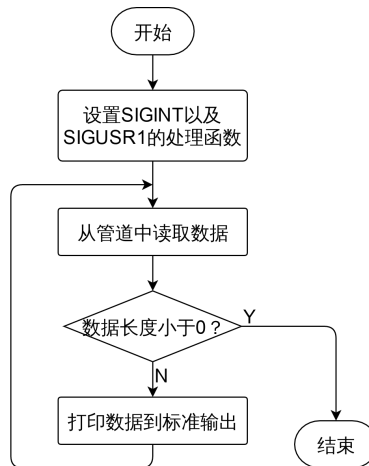


图 1.4: 子进程 2 流程

对于主进程而言，其用于处理 SIGINT 的函数 handlerMain 被触发后首先向子进程 1 和子进程 2 发送 SIGUSR1 信号，然后使用 waitpid 等待子进程 1 和子进程 2 的结束。当子进程 1 和子进程 2 结束后，打印 “Parent Process is killed”，然后使用 exit(0) 退出主进程。

而子进程 1 和子进程 2 的 SIGUSR1 处理函数则是简单的打印 “Child Process x is killed by parent!”，然后使用 exit(0) 退出进程。其中 x 为子进程号，子进程 1 的 x 为 1，子进程 2 的 x 为 2。

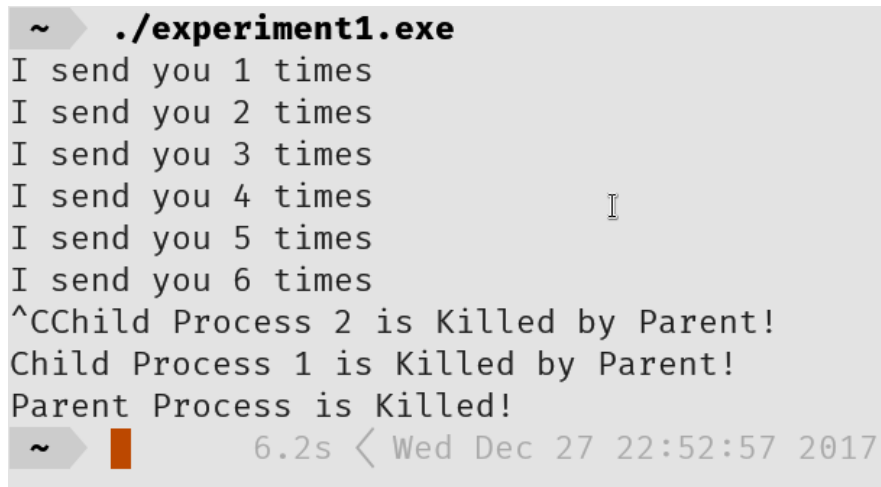
1.4 实验过程

1.4.1 开发环境

操作系统 Archlinux x86_64
编译器 gcc 7.2.1
自动编译 GNU Make 4.2.1
资源监视器 htop 2.0.2

1.4.2 实验步骤

首先编写源码，然后使用 gcc 进行编译，编译选项为 `-Wall -Werror -g -static -std=c++11 -o experiment1.exe`。编译完成后得到可执行文件 `experiment1.exe`，然后在终端下执行 `experiment1.exe`。等待若干秒之后按下 `Ctrl-C`，观察程序的输出结果，结果如图1.5所示。可以看见在 6 秒时按下了 `Ctrl-C`，在两个子进程结束后父进程结束。



```
~ > ./experiment1.exe
I send you 1 times
I send you 2 times
I send you 3 times
I send you 4 times
I send you 5 times
I send you 6 times
^CChild Process 2 is Killed by Parent!
Child Process 1 is Killed by Parent!
Parent Process is Killed!
~ > 6.2s < Wed Dec 27 22:52:57 2017
```

图 1.5: 实验一输出结果

在程序运行时打开 htop 并进行过滤，可以看到父进程和子进程的关系如1.6所示。可以看出程序确实为一个父进程 + 两个子进程的结构。在 htop 中分别将 `SIGINT` 发送给子进程 1 和子进程 2，程序依旧正常输出，没有任何变化，说明子进程能够正确的忽略掉 `SIGINT`。然后使用 `SIGUSR1` 发送给子进程 1 或子进程 2 中的任意一个，输出停止，并显示 “Child Process x is killed by parent!”，说明能够正确处理 `SIGUSR1`。最后讲 `SIGTERM` 发送给父进程，父进程显示 “Parent Process is Killed!” 后成功退出，与预期结果一致。

1.4.3 结果分析

实验过程中的程序行为与预期一致。通过标准输出可以看出程序确实能够通过管道进行通信，并能够正确的对于信号进行处理，而通过 htop 中的分别对于各个进程发送信号的结果也验证了这一点，并

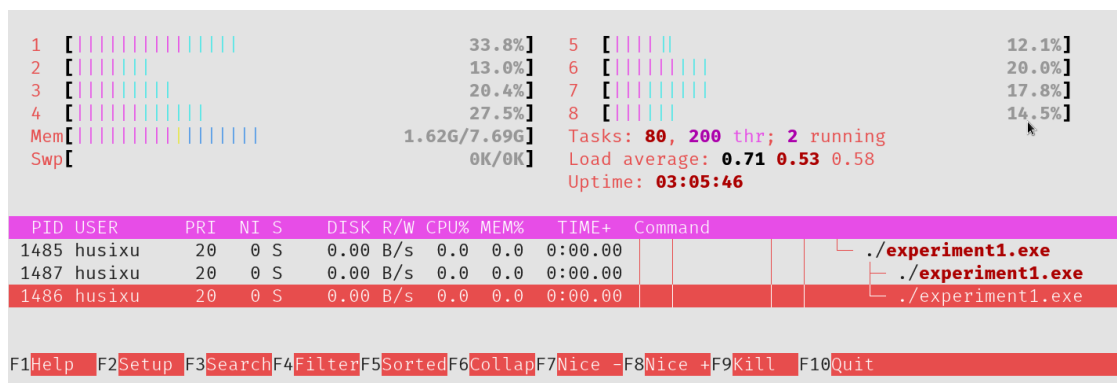


图 1.6: 进程间关系

且也说明了进程之间是能够通过信号进行同步处理的。

实验二 线程同步与通信实验

2.1 实验目的

1. 掌握 Linux 下线程的概念；
2. 了解 Linux 线程同步与通信的主要机制；
3. 通过信号灯操作实现线程间的同步与互斥。

2.2 实验内容

通过 Linux 多线程与信号灯机制，设计并实现计算机线程与 I/O 线程共享缓冲区的同步与通信。

程序要求: 两个线程, 共享公共变量 a:

线程 1 负责计算 (1 到 100 的累加, 每次加一个数)

线程 2 负责打印 (输出累加的中间结果)

2.3 实验设计

2.3.1 总体设计

在此实验的总体设计中, 总共使用两个信号量, 分别作为线程 1 计算完成信号和线程 2 打印完成信号, 分别称其为信号量 0 和信号量 1。程序的总体结构如图2.1所示。在子线程 1 在开始时首先对于

信号量 1 进行一次 P 操作每进行一次累加后作为生产者对信号量 0 进行一次 V 操作，表示可以开始打印中间结果；线程 2 在开始前先对信号量 0 进行 P 操作，输出结果后再对信号量 1 进行一次 V 操作，表示可以进行下一次累加。在累加完成后，线程 2 等待线程 1 的结束，主线程等待线程 2 的结束。

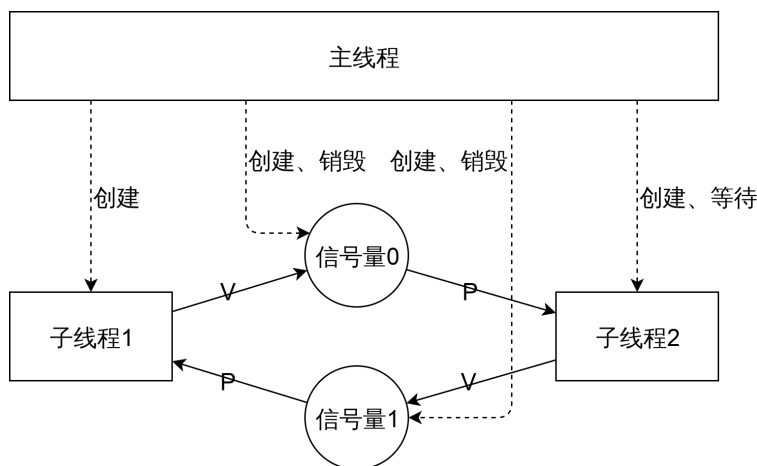


图 2.1: 程序总体结构

2.3.2 详细设计

对于主线程而言，其流程图如图1.2所示，其工作为初始化两个信号量，创建两个线程以及等待线程 2 的结束，在线程 2 结束后销毁创建的信号量并结束自身。由于线程 2 一定在线程 1 之后结束，因此主线程无需等待线程 1 的结束。在初始化信号量时，信号量 0 的初值设置为 0，信号量 1 的初值设置为 1，否则线程 1 无法进入循环，整个程序会死锁。

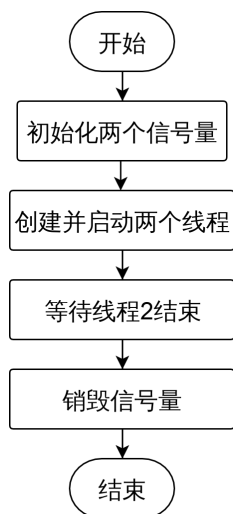


图 2.2: 主线程流程图

线程 1 的流程图如图2.3所示，在每一次循环开始时对信号量 1 进行 P 操作，在循环结束时对信号量 0 进行 V 操作。此外，在整个循环结束后，将全局变量 termFlag 设置为 1 以告知线程 2 累加操作

已经全部结束。

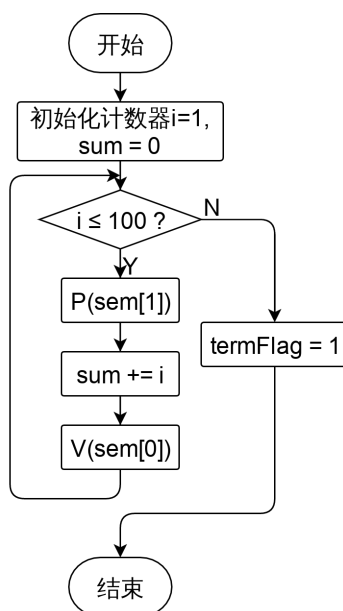


图 2.3: 线程 1 流程图

线程 2 的流程图如图2.4所示。在每次循环开始时对信号量 0 进行 P 操作，在循环结束时对信号量 1 进行 V 操作，以达到与线程 1 的同步。此外，在每次打印后需要检测线程 1 的循环是否已经完成，即 TermFlag 是否已经设置，若是，则等待线程 1 的结束并将自身结束。

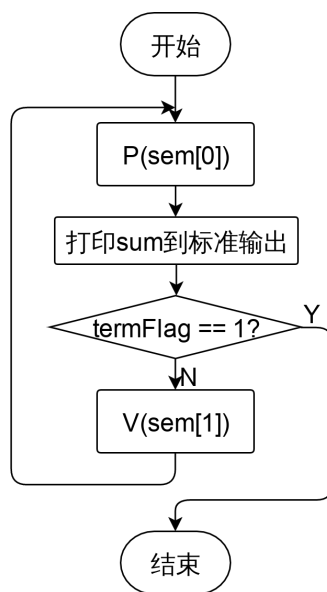


图 2.4: 线程 2 流程图

2.4 实验过程

2.4.1 开发环境

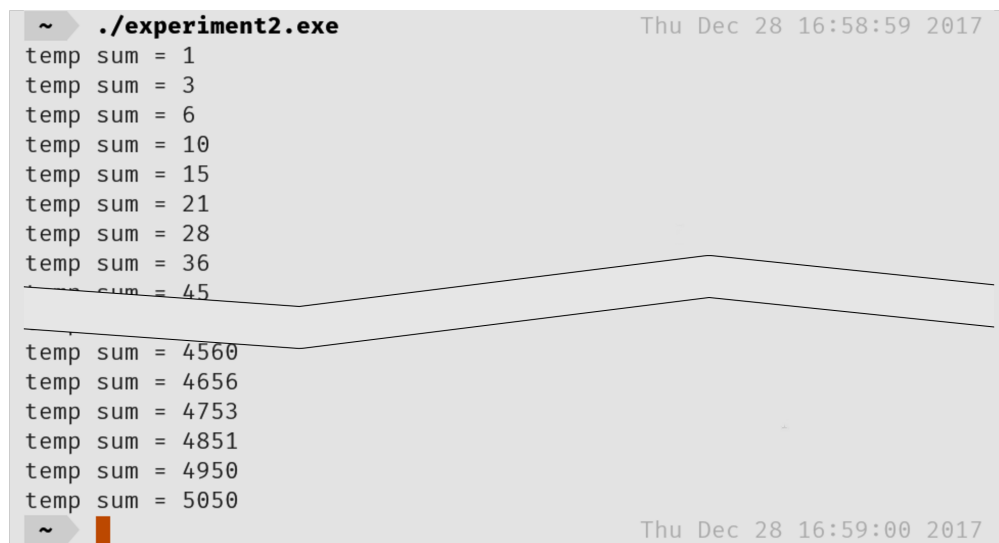
操作系统 Archlinux x86_64

编译器 gcc 7.2.1

自动编译 GNU Make 4.2.1

2.4.2 实验步骤

编写源码并编译，编译选项为-Wall -Werror -g -static -std=c++11 -o experiment2.exe，编译完成后直接运行得到的 experiment2.exe 可执行文件并观察输出结果。输出结果如图2.5所示。从结果中可以看出，程序能够正确的运行并能够正确输出每一步的结果。在输出完所有的结果后能够正确的退出，说明程序功能正常。



```
~ ./experiment2.exe Thu Dec 28 16:58:59 2017
temp sum = 1
temp sum = 3
temp sum = 6
temp sum = 10
temp sum = 15
temp sum = 21
temp sum = 28
temp sum = 36
temp sum = 45
temp sum = 4560
temp sum = 4656
temp sum = 4753
temp sum = 4851
temp sum = 4950
temp sum = 5050
~ Thu Dec 28 16:59:00 2017
```

图 2.5: 程序运行输出结果

在程序没有调试到完全正确之前，出现过输出不连续，输出完成后程序不能结束，或者完全没有输出等错误情况，经过检查后发现是由于没有正确的处理信号量的初值以及生产者-消费者的关系导致的。

2.4.3 结果分析

程序能够对于每一次迭代进行正确的输出，并且程序能够正确的退出，说明程序的逻辑是正常的。这也说明了同一程序的线程之间能够通过信号量进行同步与互斥操作。

实验三 共享内存与进程同步

3.1 实验目的

1. 掌握 Linux 下共享内存的概念与使用方法；
2. 掌握环形缓冲的结构与使用方法；
3. 掌握 Linux 下进程同步与通信的主要机制。

3.2 实验内容

利用多个共享内存（有限空间）构成的环形缓冲，将源文件复制到目标文件，实现两个进程的誊抄。

3.3 实验设计

3.3.1 总体设计

整个程序由三个进程构成：主进程、读进程与写进程。主进程用于初始化环形缓冲区、启动两个子进程并等待两个子进程的结束，读进程用于从文件中读取数据并将数据写入环形缓冲区，而写进程则负责从环形缓冲区中读取数据并写入新文件中。线程之间的同步通过三个信号量进行，分别将其称为信号量 0、1、2。信号量 0 用于标识读进程读入的数据量，信号量 1 用于标识环形缓冲中剩余的空间，信号量 2 用于进程间的互斥，即同时只允许一个进程访问环形缓冲区。整个程序的结构如3.1所示。

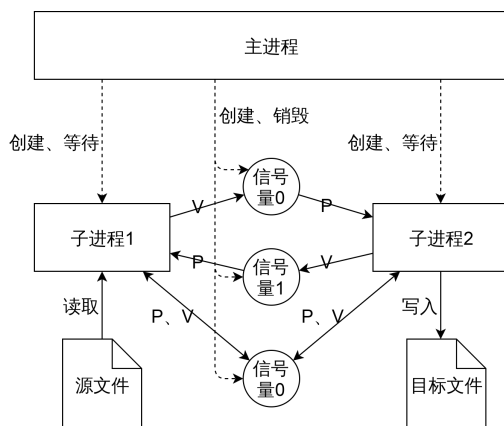


图 3.1: 程序总体结构

3.3.2 详细设计

主线程开始时首先尝试打开源文件和目标文件（文件名由命令行传入），成功后尝试创建环形缓冲区，环形缓冲区结构及其每个结点的结构如图3.2所示，每个节点中包含一个 `nextShmId` 指向下一个共享内存区，并包含一个数据区域用于存储数据。在环形缓冲区创建完成后，尝试创建 3 个信号量，由于信号量 0 用于表示读入的数据量、信号量 1 用于表示缓冲区剩余数量，信号量 2 用于加锁，因此信号量 0、1、2 的初值分别赋 0，环形缓冲区结点个数，1。最后，尝试创建读进程和写进程并等待两个进程结束。在两个进程结束后，释放所有申请的资源。在整个主线程执行的过程中，如果打开文件、创建缓冲区、创建信号量或启动子进程中的任何一步不能成功执行则立即释放已经分配的资源并退出程序。

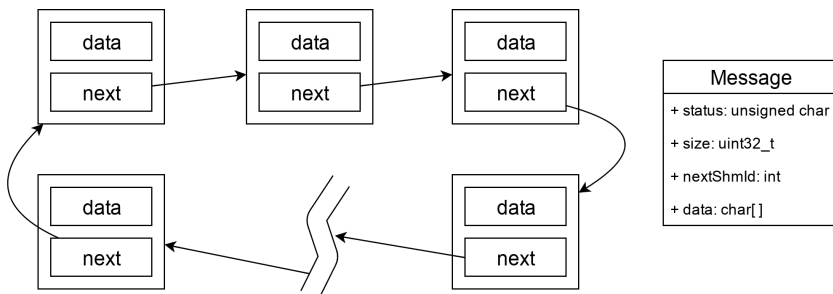


图 3.2: 缓冲区及其节点结构

读文件进程的流程图如图3.3所示，进程开始即进入读文件循环，每次循环开始前首先对信号量 1 和信号量 2 进行 P 操作，分别表示从环形缓冲区取得一个节点以及对环形缓冲区加锁。然后读取文件的一个片段，此片段大小为不大于环形缓冲区每个节点数据区的大小（由主进程在初始化环形缓冲区时指定），将其放入环形缓冲区尾部。然后判断是否读到文件尾部，若是，则关闭文件，否则执行下一个循环。在关闭文件或者执行下一个循环之前对于信号量 0 和信号量 2 进行 V 操作，表示增加一个读取的数据以及释放环形缓冲区的锁。

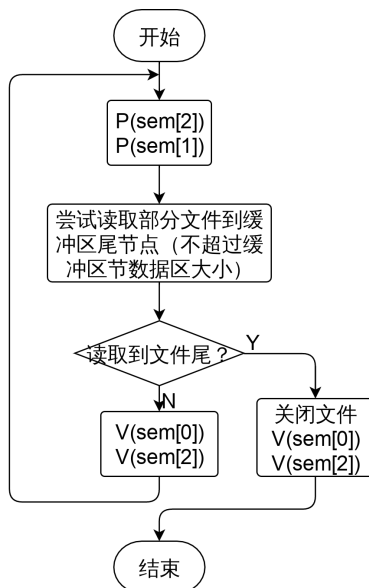


图 3.3: 读文件进程流程

写文件进程的流程与读文件进程相似，其流程图如图3.4所示。进程开始后立即进入写文件循环，每次循环开始前首先对于信号量 0 和信号量 2 进行 P 操作，表示从读取的数据片段中取一个片段以及获得环形缓冲区的锁。然后尝试将读取的数据写入文件。写入后判断这一片段是否为读文件进程向环形缓冲区添加的最后一个片段，读文件进程会向环形缓冲区节点的状态成员写入这一数据（见图3.2）。如果这一片段是最后一个片段，说明读文件进程已经完成文件读取操作，此时关闭目标文件，否则进行下一个循环。在关闭文件或进行下一个循环前对于信号量 1 和信号量 2 进行 V 操作，表示释放一个空节点到环形缓冲区，以及从释放环形缓冲区的锁。

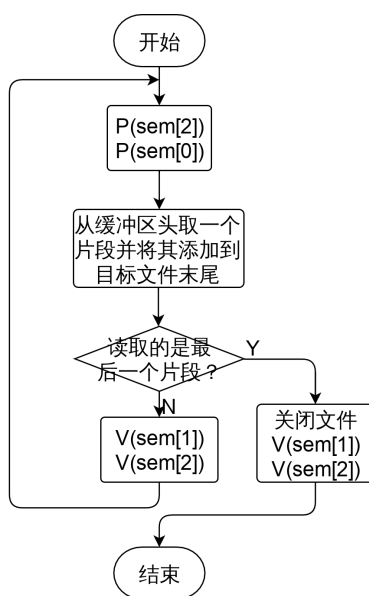


图 3.4: 写文件进程流程图

3.4 实验过程

3.4.1 开发环境

操作系统 Archlinux x86_64

编译器 gcc 7.2.1

自动编译 GNU Make 4.2.1

3.4.2 实验步骤

首先编写程序，并编译，编译选项除了目标文件外与实验一相同，得到 experiment3.exe 可执行文件。然后使用 ./experiment3.exe <src-file> <dst-file> 进行测试，其中 <src-file> 表示源文件名，<dst-file> 表示目标文件名。在本测试中，使用一个图片文件 test.png 作为源文件进行测试，测试过程如图3.5所示。从图中可以看到，首先目录下只有 experiment3.exe 以及源文件 test.png。然后使用可执行文件对源文件进行复制，并指定目标文件为 test-copy.png。在程序执行完成后，目录下出现了 test-copy.png。随后使用 md5 对于源文件和目标文件进行校验，发现源文件和目标文件是相同的，说明程序功能正常。

```
~/TestArea ls Fri Dec 29 01:19:49 2017
experiment3.exe* test.png
~/TestArea ./experiment3.exe test.png test-copy.png
~/TestArea ls 131ms < Fri Dec 29 01:19:53 2017
experiment3.exe* test-copy.png test.png
~/TestArea md5sum *.png Fri Dec 29 01:19:54 2017
119830f51c973e164c5c960f7d9f6343 test-copy.png
119830f51c973e164c5c960f7d9f6343 test.png
~/TestArea █ Fri Dec 29 01:19:55 2017
```

图 3.5: 测试过程及结果

3.4.3 结果分析

从图3.5中可以看出，文件成功地被复制并且复制前后文件的内容是相同的，说明文件复制功能正常。同时这也说明了线程之间能够通过共享内存区传递信息、以及环形缓冲区能够有效的传递信息。

实验四 Linux 文件目录

4.1 实验目的

1. 了解 Linux 文件系统与目录操作；
2. 了解 Linux 文件系统目录结构；
3. 掌握文件和目录的程序设计方法。

4.2 实验内容

编程实现目录查询功能：

功能类似 `ls -lR`；

查询指定目录下的文件及子目录信息：显示文件的类型、大小、时间等信息；

递归显示子目录中的所有文件信息。

4.3 实验设计

4.3.1 总体设计

通过 Linux 目录以及文件信息的接口，打印当前目录下的内容，并递归地调用这个程序打印这个目录所有子目录下的内容。首先研究 `ls -lR` 的输出规则，然后使用相似的规则进行输出。

4.3.2 详细设计

本程序由三个函数构成：主函数 `main`，负责递归的函数 `printDir` 以及负责打印目录和文件的函数 `printEntry`。主函数负责调用 `printDir`，`printDir` 递归调用自身并在每次递归时调用 `printDir` 打印目录下的内容。程序的逻辑较为简单但细节部分较为复杂。`ls -lR` 命令具有如下特征：

1. 首先打印传入参数所表示目录的名字以及其下的内容。若没有传入参数则默认为当前目录“.”。
2. 对于每个目录的打印，首先打印其相对路径（相对于传入参数），第二行打印 `total < 块大小 >`，且此块大小不是文件系统的块大小（解释见下），第三行以及之后的行用于打印目录下所有 Entry 的内容。
3. `ls -lR` 每个文件夹后跟的 `total` 所表示的块大小意义为： $\text{physical-blocks-in-use} \times \text{physical-block-size} \div \text{LS-BLOCK-SIZE}$ 这一参数，其中 `physical-block-size-in-use` 是文件占用的物理块的数量，`physical-block-size` 是物理块的大小，而 `LS-BLOCK-SIZE` 是可以由 `ls` 的 `--block-size` 参数指定的，若未指定则其值与 `ls` 的实现有关¹。在编写以及测试本程序的计算机上，`physical-block-size` 是 4096，`LS-BLOCK-SIZE` 是 1024，也就是说一个文件所占有的块数量至少为 4 且一定为 4 的整数倍。
4. 对于每个目录下 Entry 内容的打印顺序是根据文件名进行排序的，且是按照当前的 `locale` 设置进行排序（类似 `strcoll`）而不是直接按照字节进行排序（类似 `strcmp`）。
5. 每个 Entry 的信息打印顺序是权限-硬链接数量-所有者-拥有组-大小（字节）-修改时间-文件名。每两项之间使用一个空格隔开，每项的长度都是由所有打印出的文件的这一项的最大长度决定的（即在同一目下每一目录项的同一属性打印长度都是一致的）。除了文件名外，所有项都是右对齐的，最后修改时间包含 3 到 4 项，具体规则见下。
6. 文件的最后修改时间的打印规则是：以距离当前时间是否超过 6 个月作为分隔，超过 6 个月的只打印月份、日期以及年份。小于 6 个月的打印月份、日期、最后修改的小时数、分钟数。²

¹<https://stackoverflow.com/a/33730697>

²<https://www.gnu.org/software/coreutils/manual/coreutils.html#Formatting-file-timestamps>

4.4 实验过程

4.4.1 开发环境

操作系统 Archlinux x86_64

编译器 gcc 7.2.1

自动编译 GNU Make 4.2.1

4.4.2 实验步骤

首先编写代码，然后编译得到可执行文件 `experiment4.exe`（编译选项与实验一相同），然后对于任一目录执行命令 `./experiment4.exe < 目录名 >`，再使用 `ls -lR` 对同一个目录进行操作并对比两个程序的输出。首先对于一个简单的目录进行测试，两个程序的输出结果如图4.1所示（在 `bash` 下运行）。

```
[husixu@Ruby experiment4]$ ls -lR .
.:
total 44
-rw-r--r-- 1 husixu husixu  372 Dec 13 16:24 Makefile
-rwxr-xr-x 1 husixu husixu 20528 Jan  1 01:01 experiment4.exe
-rw-r--r-- 1 husixu husixu  8578 Dec 27 14:08 main.c
-rw-r--r-- 1 husixu husixu  1174 Dec 13 16:24 main.h
[husixu@Ruby experiment4]$ ./experiment4.exe .
.:
total 44
-rw-r--r-- 1 husixu husixu  372 Dec 13 16:24 Makefile
-rwxr-xr-x 1 husixu husixu 20528 Jan  1 01:01 experiment4.exe
-rw-r--r-- 1 husixu husixu  8578 Dec 27 14:08 main.c
-rw-r--r-- 1 husixu husixu  1174 Dec 13 16:24 main.h
[husixu@Ruby experiment4]$
```

图 4.1: 简单目录实验结果

然后对于一个较为复杂的，有层级关系的目录进行测试，直接使用 `diff` 比较两个程序的输出，如图4.2所示，可以看出，`diff` 程序直接返回而没有任何输出，说明两个程序的输出结果是一致的。

```
[husixu@Ruby ~]$ diff <(ls -lR Homework/) <(. ./experiment4.exe Homework/)
[husixu@Ruby ~]$
```

图 4.2: 复杂目录实验结果

4.4.3 结果分析

图4.1以及4.2所表示的结果表明，无论是否需要递归，`exp4Result2` 都与 `ls -lR` 具有相同的输出。另外在测试中发现，对于大型目录（如 `/home` 或根目录等）`experiment4.exe` 将由于栈溢出而崩溃，`ls -lR` 则不会，这是由于 `experient4.exe` 是直接使用递归调用函数实现的，递归层数过深时会直接导致栈溢

出，而 ls 则不是直接使用递归调用函数的方式遍历目录的。因此只能说在没有超过栈大小限制的情况下这一对目录进行操作的程序是有效的。

附录 A 实验一代码

头文件：

```
1  /**
2   * @file main.h
3   * @brief fork and pipe experiment
4   *
5   * @details
6   * this is part of the operating system course lab
7   *
8   * @date 2017-11-17
9   * @author Sixu Hu
10 **/
11 #ifndef MAIN_H_
12 #define MAIN_H_
13
14 // maximum messages
15 #define MAX_MESSAGE_SIZE 1024
16
17
18 /// @brief the child process1
19 /// this process send message to pipe to child process 2
20 /// @param[in] piped[2] the pipe
21 /// @return if the process exited normally
22 int procChild1(int piped[2]);
23
24 /// @brief the child process 2
25 /// this process receive signal from pipe from child process 1
26 /// @param[in] piped[2] the pipe
27 /// @return if the process exited normally
28 int procChild2(int piped[2]);
29
30 /// @brief the signal handler of the main process
31 /// @param sig required
32 void handlerMain(int sig);
33
```

```

34 /// @brief the signal handler of the child 1 process
35 /// @param sig required
36 void handlerChild1(int sig);
37
38 /// @brief the signal handler of the child 2 process
39 /// @param sig required
40 void handlerChild2(int sig);
41
42 #endif // MAIN_H_

```

实现文件:

```

1 #define _POSIX_SOURCE
2 #include "main.h"
3
4 #include <string.h>
5 #include <unistd.h>
6 #include <sys/wait.h>
7 #include <sys/types.h>
8 #include <signal.h>
9 #include <stdlib.h>
10 #include <stdio.h>
11
12 // the process id of two child process
13 static int pidChild1, pidChild2;
14
15 int main(int argc, char *argv[]) {
16     // register the main process signal handler
17     signal(SIGINT, handlerMain);
18
19     int piped[2];
20     // create a pipe
21     pipe(piped);
22
23     // try fork the first child
24     if ((pidChild1 = fork()) == -1)
25         return -1;
26     // run child process
27     if (pidChild1 == 0) {
28         // ignore the interrupt signal
29         signal(SIGINT, SIG_IGN);
30         // register the child 1 signal handler
31         signal(SIGUSR1, handlerChild1);
32         return procChild1(piped);
33     }
34
35     // try fork the second child

```

```
36     if ((pidChild2 = fork()) == -1)
37         return -1;
38     // run child process
39     if (pidChild2 == 0) {
40         // ignore the interrupt signal
41         signal(SIGINT, SIG_IGN);
42         // register the child 2 signal handler
43         signal(SIGUSR1, handlerChild2);
44         return procChild2(piped);
45     }
46
47     close(piped[0]);
48     close(piped[1]);
49     // close unused pipe
50     while (1) {
51         // reduce CPU occupation
52         sleep(1);
53     }
54     return 0;
55 }
56
57 int procChild1(int piped[2]) {
58     close(piped[0]);
59
60     // the message buffer
61     char buffer[MAX_MESSAGE_SIZE];
62
63     // run the main loop
64     unsigned counterSend = 1;
65     // the pipe discriptor
66     for (;;) ++counterSend) {
67         // interval: 1 second
68         sleep(1);
69         // initialize and write message to the pipe
70         sprintf(buffer, "I send you %u times", counterSend);
71         write(piped[1], buffer, strlen(buffer));
72     }
73 }
74
75 int procChild2(int piped[2]) {
76     // initialize the buffer
77     char buffer[MAX_MESSAGE_SIZE];
78     // close unused pipe
79     close(piped[1]);
80
81     ssize_t size;
```

```

82     while (1) {
83         memset(buffer, 0, MAX_MESSAGE_SIZE);
84         size = read(piped[0], buffer, MAX_MESSAGE_SIZE);
85         // handle error
86         if (size < 0)
87             return -1;
88
89         // print the message read
90         buffer[size] = 0;
91         printf("%s\n", buffer);
92     }
93 }
94
95 void handlerMain(int sig) {
96     kill(pidChild1, SIGUSR1);
97     kill(pidChild2, SIGUSR1);
98     int status;
99     waitpid(pidChild1, &status, 0);
100    waitpid(pidChild2, &status, 0);
101    printf("Parent Process is Killed!\n");
102    exit(0);
103 }
104
105 void handlerChild1(int sig) {
106     printf("Child Process 1 is Killed by Parent!\n");
107     exit(0);
108 }
109
110 void handlerChild2(int sig) {
111     printf("Child Process 2 is Killed by Parent!\n");
112     exit(0);
113 }

```

附录 B 实验二代码

头文件:

```

1  /**
2   * @file main.h
3   * @brief threas synchronize experiment

```

```

4  *
5  * @details
6  * this is part of the operating system labs
7  *
8  * @date 2017-11-17
9  * @author Sixu Hu
10 **/
11 #ifndef MAIN_H_
12 #define MAIN_H_
13
14 #include<unistd.h>
15 #include<sys/sem.h>
16
17 /// @brief routine of the first thread
18 /// This function is called when first thread started
19 /// @param[in] param the general parameter
20 /// @return a pointer to nonetype
21 void *routineThread1(void *param);
22
23 /// @brief routine of the second thread
24 /// This function is called when first thread started
25 /// @param[in] param the general parameter
26 /// @return a pointer to nonetype
27 void *routineThread2(void *param);
28
29 #endif // MAIN_H_

```

实现文件:

```

1  #define _GNU_SOURCE
2  #include"main.h"
3  #include"sem.h"
4
5  #include<pthread.h>
6  #include<unistd.h>
7  #include<stdio.h>
8  #include<sys/sem.h>
9  #include<sys/stat.h>
10 #include<errno.h>
11 #include<stdlib.h>
12 #include<limits.h>
13
14
15 // the global semaphore
16 static int semId;
17 // the sum
18 static int sum = 0;

```

```
19 // two threads
20 static pthread_t thread1, thread2;
21 // termination flag
22 static int terminateFlag = 0;
23
24 int main() {
25     // initialize semaphore
26     key_t semkey;
27     if ((semkey = ftok("/tmp", 'a')) == (key_t)(-1)) {
28         perror("IPC error: ftok");
29         exit(1);
30     }
31     semId = create_Sem(semkey, 2);
32     set_N(semId, 0, 0);
33     set_N(semId, 1, 1);
34
35     // create two thread
36     pthread_create(&thread1, NULL, routineThread1, NULL);
37     pthread_create(&thread2, NULL, routineThread2, NULL);
38
39     // wait for threads to finish
40     pthread_join(thread2, NULL);
41
42     // destroy semaphore
43     destroy_Sem(semId);
44     return 0;
45 }
46
47 void *routineThread1(void *param) {
48     // calculate the sum from 1 to 100
49     for (int i = 1; i <= 100; ++i) {
50         P(semId, 1);
51         sum += i;
52         V(semId, 0);
53     }
54     terminateFlag = 1;
55     return NULL;
56 }
57
58 void *routineThread2(void *param) {
59     while (1) {
60         P(semId, 0);
61         printf("temp sum = %d\n", sum);
62
63         // try to wait for thread 1
64         if (terminateFlag) {
```

```

65         pthread_join(thread1, NULL);
66         return NULL;
67     }
68
69     V(semId, 1);
70 }
71 return NULL;
72 }

```

附录 C 实验三代码

头文件:

```

1  #ifndef MAIN_H_
2  #define MAIN_H_
3
4  #include<stdio.h>
5  #include<stdint.h>
6
7  #define BUFFERNUM 1024
8
9  /// @brief the shared memory data size (in bytes, not including control messages)
10 /// This value must be less than (2 ^ 31 - 1)
11 #define DATASIZE 1000
12
13 #define STATUS_PENDING  0x01
14 #define STATUS_READ     0x02
15 #define STATUS_WRITTEN  0x04
16 #define STATUS_ALL      0x08
17
18 #define STATUS_HEAD     0x10
19
20 #define STATUS_TAIL     0x20
21
22 /// @brief the message structure
23 typedef struct _Message {
24     unsigned char status;    //!< current shared buffer status
25     uint32_t size;          //!< current message size
26     int nextShmId;          //!< pointer to the next Message
27     char data[DATASIZE];    //!< where data is stored

```

```

28 } Message;
29
30 /// @brief the size of buffer header (used to store read/write status and bytes read)
31 /// |<- status 1byte ->|<- size 4bytes ->|
32 #define SIZE_HEADER 5
33
34 /// @brief the reader's process
35 /// @param[in] inFile the input file descriptor
36 /// @param[in] idShmHead the head shared memory id of the buffer
37 /// @param[in] idShmTail the tail shared memory id of the buffer
38 /// @param[in] idSem the semaphore id
39 /// @return return value, 0 for success
40 int procRead(FILE *inFile, int idShmHead, int idShmTail, int idSem);
41
42 /// @brief the writer's process
43 /// @param[in] outFile the output file descriptor
44 /// @param[in] idShmHead the head shared memory id of the buffer
45 /// @param[in] idShmTail the tail shared memory id of the buffer
46 /// @param[in] idSem the semaphore id
47 /// @return return value, 0 for success
48 int procWrite(FILE *outFile, int idShmHead, int idShmTail, int idSem);
49
50 #endif // MAIN_H_

```

实现文件:

```

1 #define _GNU_SOURCE
2 #include "../common/log.c/src/log.h"
3
4 #include "main.h"
5 #include "sem.h"
6
7 #include <stdio.h>
8 #include <string.h>
9 #include <unistd.h>
10 #include <sys/shm.h>
11 #include <stdlib.h>
12 #include <signal.h>
13 #include <sys/wait.h>
14
15 // the process id
16 static int idProcRead = 0, idProcWrite = 0;
17
18 int main(int argc, char *argv[]) {
19     // display hint
20     if (argc != 3) {
21         printf("Usage: %s <source-file> <dst-file>", argv[0]);

```



```

22     return 1;
23 }
24
25 // check files' validity
26 FILE *fileIn;
27 FILE *fileOut;
28 if ((fileIn = fopen(argv[1], "rb")) == NULL) {
29     perror("failed to open file for reading.");
30     return 1;
31 }
32 if ((fileOut = fopen(argv[2], "wb")) == NULL) {
33     perror("failed to open file for writing,");
34     return 1;
35 }
36
37 // get a unique key
38 key_t shmkey;
39 if ((shmkey = ftok("./", 'a')) == (key_t)(-1)) {
40     perror("IPC error: ftok");
41     exit(1);
42 }
43 // create the shared memory and initialise it
44 int idHeadShm;
45 if ((idHeadShm = shmget(shmkey, sizeof(Message), IPC_CREAT | 0660)) <= 0) {
46     perror("Failed to create shared memory");
47     exit(1);
48 }
49 Message *messageTemp = (Message *)shmat(idHeadShm, NULL, 0);
50 if ((int64_t)(messageTemp) == -1) {
51     perror("Failed to retrieve shared memory");
52     exit(1);
53 }
54 messageTemp->status = STATUS_PENDING | STATUS_HEAD | STATUS_TAIL;
55
56 // create the circle buffer
57 for (int i = 0; i < BUFFERNUM; ++i) {
58     // get a unique key
59     key_t shmkey;
60     if ((shmkey = ftok("./", i)) == (key_t)(-1)) {
61         perror("IPC error: ftok");
62         exit(1);
63     }
64     // create the shared memory and initialise it
65     int idShm;
66     if ((idShm = shmget(shmkey, sizeof(Message), IPC_CREAT | 0660)) <= 0) {
67         perror("Failed to create shared memory");

```

```
68         exit(1);
69     }
70     messageTemp->nextShmId = idShm;
71     messageTemp->status = STATUS_PENDING;
72     messageTemp = (Message *)shmat(idShm, NULL, 0);
73     if ((int64_t)(messageTemp) == -1) {
74         perror("Failed to retrieve shared memory");
75         exit(1);
76     }
77 }
78 messageTemp->nextShmId = idHeadShm;
79
80 // get a unique key
81 key_t semKey;
82 if ((semKey = ftok("./", 'b')) == (key_t)(-1)) {
83     perror("IPC error: ftok");
84     exit(1);
85 }
86 // create lock for PV operation
87 int idSem;
88 if ((idSem = create_Sem(semKey, 4)) < 0) {
89     perror("Failed to create semaphore");
90     exit(1);
91 }
92 set_N(idSem, 0, 0);
93 set_N(idSem, 1, BUFFERNUM-1);
94 set_N(idSem, 2, 1);
95
96 // create read process
97 if ((idProcRead = fork()) == -1) {
98     perror("Failed to create process.");
99     return 1;
100 }
101 // run the reader's process
102 if (idProcRead == 0)
103     return procRead(fileIn, idHeadShm, idHeadShm, idSem);
104
105 // create write process
106 if ((idProcWrite = fork()) == -1) {
107     perror("Failed to create process.");
108     // clean former threads
109     kill(idProcRead, SIGKILL);
110     return 1;
111 }
112 // run the writer's process
113 if (idProcWrite == 0)
```

```
114     return procWrite(fileOut, idHeadShm, idHeadShm, idSem);
115
116     // wait two process to stop
117     int status;
118     waitpid(idProcRead, &status, 0);
119     waitpid(idProcWrite, &status, 0);
120
121     destroy_Sem(idSem);
122     return 0;
123 }
124
125 int procRead(FILE *inFile, int idShmHead, int idShmTail, int idSem) {
126     // start to read file to buffer
127     Message *messageTail = (Message *)shmat(idShmTail, NULL, 0);
128
129     while (1) {
130         P(idSem, 2);
131         P(idSem, 1);
132         // read if quque is not full
133         size_t bytesRead;
134         if ((bytesRead = fread((void *) (messageTail->data), 1, DATASIZE, inFile)) ==
135             0) {
136             // if file read finished
137             messageTail->status = STATUS_ALL;
138             messageTail->size = bytesRead;
139             fclose(inFile);
140             V(idSem, 0);
141             V(idSem, 2);
142             return 0;
143         }
144         messageTail->size = bytesRead;
145
146         // move tail to next node
147         idShmTail = messageTail->nextShmId;
148         messageTail = (Message *)shmat(idShmTail, NULL, 0);
149
150         V(idSem, 0);
151         V(idSem, 2);
152     }
153
154 int procWrite(FILE *outFile, int idShmHead, int idShmTail, int idSem) {
155     Message *messageHead = (Message *)shmat(idShmHead, NULL, 0);
156     while (1) {
157         P(idSem, 2);
158         P(idSem, 0);
```

```

159         // if all the messages are read from file
160         if (messageHead->status == STATUS_ALL) {
161             fwrite((void*)(messageHead->data), messageHead->size, 1, outFile);
162             fclose(outFile);
163             V(idSem, 1);
164             V(idSem, 2);
165             return 0;
166         }
167
168         fwrite((void*)(messageHead->data), messageHead->size, 1, outFile);
169         // clear current node and move head to next
170         idShmHead = messageHead->nextShmId;
171         messageHead = (Message *)shmat(idShmHead, NULL, 0);
172         V(idSem, 1);
173         V(idSem, 2);
174     }
175 }

```

附录 D 实验四代码

头文件:

```

1 /**
2  * @file main.h
3  * @brief An experiment trys to reimplement ls -lR
4  *
5  * @details
6  * This program is expected to have the exactly same output with ls -lR,
7  * note that ls -lR behaves differently on different system/machines so
8  * maybe the outputs are not exactly the same (I'll try my best).
9  *
10 * @date 2017-11-23
11 * @author Sixu Hu
12 **/
13 #ifndef MAIN_H_
14 #define MAIN_H_
15
16 #include<dirent.h>
17 #include<time.h>
18

```

```

19 /// @brief Print directories recursively rooted at dir
20 /// @param[in] dir The root directory to print
21 /// @param[in] depth Max depth to print
22 int printDir(char *dir, int depth);
23
24 /// @brief Print one directory entry
25 /// @param[in] entry The entry to print
26 /// @param[in] fmtLinkWidth The width of link in displaying
27 /// @param[in] fmtSizeWidth The width of file size in displaying
28 /// @return 0 if print is successful
29 int printEntry(struct dirent *entry, int fmtLinkWidth, int fmtSizeWidth);
30
31 /// @brief The compare function used to sort two directory entries
32 /// @param a One dir entry
33 /// @param b Another dir entry
34 /// @return An int indicating the order.
35 int dirEntrySortCompare(const void *a, const void *b);
36
37 #define LS_BLOCK_SIZE 1024
38
39 #endif // MAIN_H_

```

实现文件:

```

1 /**
2  * @file main.c
3  * @brief An experiment trys to reimplement ls -lR
4  *
5  * @details
6  * This program is expected to have the exactly same output with ls -lR,
7  * note that ls -lR behaves differently on different system/machines so
8  * maybe the outputs are not exactly the same (I'll try my best).
9  *
10 * @date 2017-11-23
11 * @author Sixu Hu
12 */
13 #define _DEFAULT_SOURCE
14 #include "main.h"
15
16 #include <sys/stat.h>
17 #include <sys/types.h>
18 #include <sys/cdefs.h>
19 #include <bsd/string.h>
20
21 #include <grp.h>
22 #include <pwd.h>
23 #include <dirent.h>

```

```
24 #include<libgen.h>
25 #include<unistd.h>
26 #include<stdio.h>
27 #include<stdlib.h>
28 #include<string.h>
29 #include<time.h>
30 #include<locale.h>
31
32 int main(int argc, char *argv[]) {
33     if (argc != 2) {
34         printf("Usage %s <directory-or-file>\n", argv[0]);
35         return 1;
36     }
37
38     /// set system locale according to
39     /// \ [this](https://www.gnu.org/software/libc/manual/html_node/Setting-the-Locale.html)
40     /// \ for correct sorting order using the system locale sort
41     setlocale(LC_COLLATE, "");
42
43     // start printing directory
44     char *path = malloc(strlen(argv[1]) + 1);
45     strcpy(path, argv[1]);
46     printDir(path, 0);
47     free(path);
48 }
49
50 static time_t now;
51
52 int printDir(char *dir, int depth) {
53     // set current time
54     now = time(NULL);
55
56     // backup the old directory
57     char *dirOld = NULL;
58     long currentPathSize = pathconf(".", _PC_PATH_MAX);
59     dirOld = getcwd(dirOld, currentPathSize);
60
61     // get current directory status
62     /// @quote Both dirname() and basename() may modify the contents of path,
63     /// \ so it may be desirable to pass a copy when calling one of these functions.
64     /// — ‘man dirname.3’
65     char *dirTemp = malloc(strlen(dir) + 1);
66     strcpy(dirTemp, dir);
67     char *dirBase = basename(dirTemp);
68     struct stat statusTemp;
```

```

69     stat(dirBase, &statusTemp);
70     __blkcnt_t blockTotalCount = 0;
71
72     // set current directory
73     DIR *dirCurrent = NULL;
74     // judge if absolute path
75     if (dir[0] == '/') {
76         if ((dirCurrent = opendir(dir)) == NULL) {
77             perror("Failed to open directory");
78             return 1;
79         }
80         chdir(dir);
81     } else {
82         if ((dirCurrent = opendir(dirBase)) == NULL) {
83             perror("Failed to open directory");
84             return 1;
85         }
86         chdir(dirBase);
87     }
88
89     // print current directory (following the upper directory)
90     printf("%s:\n", dir);
91
92     ////////////////////////////////////
93     // Read All entries //
94     ////////////////////////////////////
95
96     // all entries in the current directory
97     struct dirent *entryItems;
98     // read entry nums
99     struct dirent *entryTemp;
100    size_t entryCount = 0;
101    while ((entryTemp = readdir(dirCurrent)))
102        entryCount++;
103
104    // read all entries, add to entryItems
105    entryItems = (struct dirent *)malloc(entryCount * sizeof(struct dirent));
106    rewinddir(dirCurrent);
107    for (int i = 0; (entryTemp = readdir(dirCurrent)); ++i)
108        memcpy(entryItems + i, entryTemp, sizeof(struct dirent));
109
110    // sort all entries
111    qsort(entryItems, entryCount, sizeof(struct dirent), dirEntrySortCompare);
112
113    ////////////////////////////////////
114    // Process current directory //

```

```

115  //////////////////////////////////////
116
117  // get max link width, max size with, total block size,
118  // and gather some other informations for formatting and printing
119  __nlink_t maxLinkNum = 0;
120  __off_t maxSize = 0;
121  int sizeWidth = 0, linkWidth = 0;
122  struct stat status;
123  for (int i = 0; i < entryCount; ++i) {
124      entryTemp = entryItems + i;
125      stat(entryTemp->d_name, &status);
126      // update max size
127      if (status.st_size > maxSize)
128          maxSize = status.st_size;
129
130
131      // update total block size
132      if (entryTemp->d_name[0] != '.' &&                // pass current file
133          strcmp(entryTemp->d_name, ".") &&            // pass upper file
134          strcmp(entryTemp->d_name, "..") ) {          // pass hidden file
135          blockTotalCount += status.st_blocks;
136          // update max link-num
137          if (status.st_nlink > maxLinkNum)
138              maxLinkNum = status.st_nlink;
139      }
140  }
141  // calculate widths
142  for (; maxLinkNum; maxLinkNum /= 10)
143      ++linkWidth;
144  for (; maxSize; maxSize /= 10)
145      ++sizeWidth;
146
147  /// print block size
148  /// according to [this](https://stackoverflow.com/a/33730697) and 'man stat.2'
149  printf("total %lu\n", blockTotalCount * 512 / LS_BLOCK_SIZE );
150
151  // read and print currnet entries
152  rewinddir(dirCurrent);
153  for (int i = 0; i < entryCount; ++i) {
154      entryTemp = entryItems + i;
155      // skip current and upper directory
156      if (strcmp(entryTemp->d_name, ".") &&                // pass current file
157          strcmp(entryTemp->d_name, "..") &&            // pass upper file
158          entryTemp->d_name[0] != '.')                // pass hidden file
159          printEntry(entryTemp, linkWidth, sizeWidth);
160  }

```



```

161
162     ///////////////
163     // Recursion //
164     ///////////////
165
166     // recurse into deeper directories
167     rewinddir(dirCurrent);
168     for (int i = 0; i < entryCount; ++i) {
169         entryTemp = entryItems + i;
170         if (entryTemp->d_type == DT_DIR &&
171             entryTemp->d_name[0] != '.' &&           // pass current file
172             strcmp(entryTemp->d_name, ".") &&         // pass upper file
173             strcmp(entryTemp->d_name, "..") ) {       // pass hidden file
174
175             // construct path
176             char *dirTemp = (char *)malloc(strlen(dir) + strlen(entryTemp->d_name) +
177                                     2);
178             strcpy(dirTemp, dir);
179             if (dir[strlen(dir) - 1] != '/')
180                 strcat(dirTemp, "/");
181             strcat(dirTemp, entryTemp->d_name);
182
183             // recurse into next directory
184             putchar('\n');
185             printDir(dirTemp, depth + 1);
186             free(dirTemp);
187         }
188     }
189
190     // clean resources and go back to the upper directory
191     closedir(dirCurrent);
192     chdir(dirOld);
193     free(dirOld);
194     free(entryItems);
195     return 0;
196 }
197
198 int printEntry(struct dirent *entry, int fmtLinkWidth, int fmtSizeWidth) {
199     struct stat status;
200
201     // read and print status
202     if (stat(entry->d_name, &status) == 0) {
203         // get permission (strmode will append a space after the string)
204         char strMode[13];
205         strmode(status.st_mode, strMode);

```

```

206 // get username and groupname
207 struct passwd *passwdTemp;
208 struct group *groupTemp;
209 /// @quote The return value may point to a static area,
210 /// \ and may be overwritten by subsequent calls to getpwent(3),
211 /// \ getpwnam(), or getpwuid(). (Do not pass the returned pointer to free(3)
    .)
212 /// — 'man getpwuid'
213 passwdTemp = getpwuid(status.st_uid);
214 /// @quote The return value may point to a static area,
215 /// \ and may be overwritten by subsequent calls to getpwent(3),
216 /// \ getpwnam(), or getpwuid(). (Do not pass the returned pointer to free(3)
    .)
217 /// — 'man getgrgid'
218 groupTemp = getgrgid(status.st_gid);
219
220 // convert time to redable format
221 /// @quote A timestamp is considered to be recent if it is less than six
    months old, and is not dated in the future. If a timestamp dated today is
    not listed in recent form, the timestamp is in the future, which means you
    probably have clock skew problems which may break programs like make that
    rely on file timestamps.
222 /// — from [here](https://www.gnu.org/software/coreutils/manual/coreutils.
    html#Formatting-file-timestamps)
223 struct tm *timeTemp = localtime(&(status.st_mtime));
224 char strTime[20];
225 // six month tricks
226 if (now - status.st_mtime > (365 * 24 * 60 * 60 / 2) || now < status.st_mtime)
    {
227     strftime(strTime, 20, "%b %d %Y", timeTemp);
228 } else {
229     strftime(strTime, 20, "%b %d %H:%M", timeTemp);
230 }
231
232 // print permission, hardlink-number, username, groupe name, size, modify-date
    , name
233 printf("%s*ld %s %s %*ld %s %s\n",
234         strMode,
235         fmtLinkWidth, status.st_nlink,
236         passwdTemp->pw_name,
237         groupTemp->gr_name,
238         fmtSizeWidth, status.st_size,
239         strTime,
240         entry->d_name);
241 return 0;
242 }

```

```
243     return 1;
244 }
245
246 int dirEntrySortCompare(const void *a, const void *b) {
247     // the two directory
248     struct dirent *dirA = (struct dirent *) a,
249                     *dirB = (struct dirent *) b;
250
251     // sort according to current locale
252     return strcoll(dirA->d_name, dirB->d_name);
253 }
```