

西安交通大学

操作系统专题实验报告

班级:  _____

学号:  _____

姓名:  _____

2022 年 12 月 15 日

目录

1 openEuler 系统环境实验.....	1
1.1 实验目的	1
1.2 实验内容	1
1.3 实验思想	2
1.4 实验步骤	2
1.5 测试数据设计	3
1.6 程序运行初值及运行结果分析	3
1.7 实验总结	7
1.7.1 实验中的问题与解决过程	7
1.7.2 实验收获	7
1.7.3 意见与建议	7
1.8 附件	8
1.8.1 附件 1 程序	8
1.8.2 附件 2 Readme	12
2 进程通信与内存管理	17
2.1 实验目的	17
2.2 实验内容	17
2.3 实验思想	18
2.4 实验步骤	18
2.5 测试数据设计	18
2.6 程序运行初值及运行结果分析	19
2.7 页面置换算法复杂度分析	23
2.8 回答问题	24
2.8.1 软中断通信	24
2.8.2 管道通信	25
2.9 实验总结	26
2.9.1 实验中的问题与解决过程	26
2.9.2 实验收获	26
2.9.3 意见与建议	26
2.10 附件	27

2.10.1 附件 1 程序.....	27
2.10.2 附件 2 Readme.....	35
3 文件系统.....	42
3.1 实验目的	42
3.2 实验内容	42
3.3 实验思想.....	42
3.4 实验步骤.....	42
3.5 实验运行初值及运行结果分析.....	46
3.6 实验总结	50
3.6.1 实验中的问题与解决过程.....	50
3.6.2 实验收获	52
3.6.3 意见与建议	52
3.7 附件	52
3.7.1 附件 1 程序	52
3.7.2 附件 2 Readme	72

1 openEuler 系统环境实验

1.1 实验目的

- 1) 熟悉基于 Kunpeng 架构弹性云服务器 ECS 上 openEuler 操作系统的基本系统环境;
- 2) 熟悉操作命令、编辑、编译、运行程序;
- 3) 了解进程调度的过程、孤儿进程和僵尸进程、物理地址与虚地址概念;
- 4) 体会线程共享进程信息、线程对共享变量操作中同步与互斥的知识。

1.2 实验内容

(一) 进程实验

- 1) 完成操作系统原理课程教材 P103 习题 3.7 (书中图 3-32 所示的程序) 的运行验证, 多运行程序几次观察结果, 去除 wait 后再观察结果并进行理论分析;
- 2) 扩展如上所述的程序:
 - a) 添加一个全局变量并在父进程和子进程中对这个变量做不同操作, 输出操作结果并解释, 同时输出两种变量的地址, 观察并分析;
 - b) 在 return 前增加对全局变量的操作并输出结果, 观察并解释;
 - c) 修改程序体会在子进程中调用 system 函数和在子进程中调用 exec 族函数执行自己写的一段程序, 在此程序中输出进程 PID 进行比较并分析。

(二) 线程实验

- 1) 在进程中给一变量赋初值并创建两个线程;
- 2) 在两个线程中分别对此变量循环五千次以上做不同的操作并输出结果;
- 3) 多运行几遍程序观察运行结果, 如果发现每次运行结果不同, 请解释原因并修改程序解决, 考虑如何控制互斥和同步;
- 4) 将进程实验中调用 system 函数和调用 exec 族函数改成在线程中实现, 观察运行结果, 输出进程 PID 与线程 TID 进行比较并分析。

1.3 实验思想

通过华为云平台，了解国产操作系统 openEuler，并熟悉一些基本的命令行操作（如在命令行中使用 gcc 编译 c 源程序等）。

通过进程与线程的实验，将理论与实践相结合，理解并掌握一些基本的系统调用函数（如：fork、wait、system 等），以及 Pthreads API 的使用。同时，熟悉进程与线程的创建与使用方法，并明确二者的区别与联系。

1.4 实验步骤

（一）进程实验

- 1) 多次运行图 1-1 中的程序，去掉 wait 函数后再次运行，观察每次的运行结果并进行分析；

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid, pid1;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        pid1 = getpid();
        printf("child: pid = %d",pid); /* A */
        printf("child: pid1 = %d",pid1); /* B */
    }
    else { /* parent process */
        pid1 = getpid();
        printf("parent: pid = %d",pid); /* C */
        printf("parent: pid1 = %d",pid1); /* D */
        wait(NULL);
    }

    return 0;
}
```

图 1-1 教材 P103 习题 3.7 中的程序

- 2) 初始化全局变量值为 0，在父进程中对其执行递增操作，在子进程中对其执行递减操作，在父、子进程中分别输出全局变量的值和地址。在程序 return 前递增全局变量，然后输出全局变量的值和地址；

- 3) 在父、子进程中分别输出进程 PID，并在子进程中调用 system 函数或 exec 族函数执行自己编写的一段程序（该程序打印“Hello World”，并输出进程 PID）。

(二) 线程实验

- 1) 初始化全局变量值为 0，创建两个线程，分别对全局变量进行 5000 次递增和递减操作，并输出全局变量的值；
- 2) 使用互斥锁修改程序，控制线程的同步与互斥；
- 3) 在线程中调用 system 函数或 exec 族函数执行自己编写的一段程序（该程序打印“Hello World”）。同时，在进程、线程及自编程序中使用 syscall(SYS_gettid)、getpid()、pthread_self() 输出进程（线程）ID。

1.5 测试数据设计

调用 system 函数或 exec 族函数执行自己编写的程序如下：

```
//system或exec族函数调用的程序，可执行文件名为hello
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/syscall.h>

int main() {
    printf("Hello World!\n");
    //printf("hello: pid = %d\n", getpid());           //进程实验中使用
    printf("hello: syscall(SYS_gettid) = %ld, getpid() = %d, pthread_self() = %ld\n",
           syscall(SYS_gettid), getpid(), pthread_self()); //线程实验中使用
    return 0;
}
```

1.6 程序运行初值及运行结果分析

(一) 进程实验

- 1) wait 函数

图 1-1 中的程序运行结果如下：

```
[root@kp-test01 test]# ./pid
child: pid = 0child: pid1 = 2391parent: pid = 2391parent: pid1 = 2390[root@kp-test01 test]# ./pid
child: pid = 0child: pid1 = 2417parent: pid = 2417parent: pid1 = 2416[root@kp-test01 test]# ./pid
child: pid = 0child: pid1 = 2431parent: pid = 2431parent: pid1 = 2430[root@kp-test01 test]# ./pid
child: pid = 0child: pid1 = 2451parent: pid = 2451parent: pid1 = 2450[root@kp-test01 test]# ./pid
child: pid = 0child: pid1 = 2465parent: pid = 2465parent: pid1 = 2464[root@kp-test01 test]# ./pid
child: pid = 0child: pid1 = 2479parent: pid = 2479parent: pid1 = 2478[root@kp-test01 test]# ./pid
child: pid = 0child: pid1 = 2493parent: pid = 2493parent: pid1 = 2492[root@kp-test01 test]# ./pid
child: pid = 0child: pid1 = 2507parent: pid = 2507parent: pid1 = 2506[root@kp-test01 test]# ./pid
```

图 1-2 带 wait 函数

修改程序，将 wait 函数去掉后，程序重新编译运行的结果如下：

```
[root@kp-test01 test]# ./pid
parent: pid = 2570parent: pid1 = 2569child: pid = 0child: pid1 = 2570[root@kp-test01 test]# ./pid
parent: pid = 2590parent: pid1 = 2589child: pid = 0child: pid1 = 2590[root@kp-test01 test]# ./pid
child: pid = 0child: pid1 = 2610parent: pid = 2610parent: pid1 = 2609[root@kp-test01 test]# ./pid
parent: pid = 2624parent: pid1 = 2623child: pid = 0child: pid1 = 2624[root@kp-test01 test]# ./pid
parent: pid = 2638parent: pid1 = 2637child: pid = 0child: pid1 = 2638[root@kp-test01 test]# ./pid
parent: pid = 2652parent: pid1 = 2651child: pid = 0child: pid1 = 2652[root@kp-test01 test]# ./pid
parent: pid = 2666parent: pid1 = 2665child: pid = 0child: pid1 = 2666[root@kp-test01 test]# ./pid
parent: pid = 2680parent: pid1 = 2679child: pid = 0child: pid1 = 2680[root@kp-test01 test]# ./pid
```

图 1-3 去掉 wait 函数

结果分析：

在两种情况下，子进程的 pid 总是为 0，且子进程的 pid1 与父进程的 pid 始终相等，父进程的 pid 始终比其 pid1 大 1。这恰恰体现出了 fork 函数返回值的特点：在父进程中，fork 返回新创建子进程的进程 ID；在子进程中，fork 返回 0；如果出现错误，fork 返回 -1。

同时，观察发现：原本的程序输出都是子进程先于父进程，但在去掉 wait 函数后，父、子进程的输出先后次序就不一定了。问题的关键在于 printf 函数的输出缓冲区：只有在缓冲区满、出现换行符、进程结束等情况下，printf 函数才会刷新缓冲区，将缓冲区中的内容显示在屏幕上。而在原本的程序中，父进程调用 wait 函数阻塞自己以等待子进程结束，故子进程会先于父进程结束，自然子进程就会比父进程先输出。去掉 wait 函数后，父、子进程哪个先结束取决于 CPU 调度，因而两者的输出次序并不一定。

2) 全局变量

程序运行结果如下：

```
[root@kp-test01 test]# ./global
child: count = -1
child: count address = 0x420054
parent: count = 1
parent: count address = 0x420054
```

图 1-4 全局变量

观察发现：子进程中 count 值为 -1，父进程中 count 值为 1，但两者地址相同。

结果分析：

调用 fork 函数后，子进程完全复制父进程的地址空间，也复制了页表，但没有复制物理页面，所以这时虚拟地址相同，物理地址也相同，但是会把父、子进程共享的页面标记为“只读”。只有当其中任何一个进程要对共享的页面执行“写操作”时，内核才会复制一个物理页面给这个进程使用，同时修改页表，并把原来的只读页面标记为“可写”，留给另外一个进程使用。这就是所谓的“写时复制”技术。

因此，本实验中父、子进程的 count 变量虽然虚拟地址相同，但实际的物理地址不同，因而才有了两个不同的值。

在 return 前递增全局变量后，输出全局变量的值和地址，程序运行结果如下：

```
[root@kp-test01 test]# ./global
child: count = -1
child: count address = 0x420054
before return: count = 0
before return: count address = 0x420054
parent: count = 1
parent: count address = 0x420054
before return: count = 2
before return: count address = 0x420054
```

图 1-5 return 前递增全局变量

观察发现：在程序 return 前递增全局变量后，子进程中 count 值为 0，父进程中 count 值为 2，且两者地址相同。

结果分析：同样是由于父、子进程中全局变量的虚拟地址相同但物理地址不同，因此子进程中原本为-1 的 count 递增为 0，而父进程中原本为 1 的 count 递增为 2。

3) system 函数与 exec 族函数

调用 system 函数的运行结果如下：

```
[root@kp-test01 test]# ./p_system
child: pid = 2433
Hello World!
hello: pid = 2434
parent: pid = 2432
```

图 1-6 进程中调用 system 函数

调用 execl 函数的运行结果如下：

```
[root@kp-test01 test]# ./p_execl
child: pid = 2508
Hello World!
hello: pid = 2508
parent: pid = 2507
```

图 1-7 进程中调用 execl 函数

这里只调用了 exec 族函数之一，其它 5 个函数(execlp、execle、execv、execvp、execvpe)的作用也与此类似，故不再一一演示。

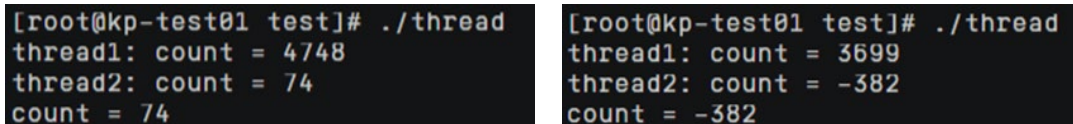
观察发现：调用 system 后 hello 程序中的 pid 比子进程 pid 大 1，而调用 execl 后 hello 程序中的 pid 与子进程相同。

结果分析：调用 system 函数会再创建一个子进程，并在该子进程中执行参数中的命令，因此它拥有独立的代码空间及数据空间，且等待新的进程执行完毕，system 函数才会返回；而当进程调用一种 exec 族函数时，该进程的用户空间代码和数据完全被新程序替换，从新程序的启动例程开始执行（当前进程剩下的程序代码便不再执行）。调用 exec 族函数并不创建新进程，所以调用 execl 函数前后该进程的 ID 并未改变。

(二) 线程实验

1) 同步与互斥

未加锁前，程序运行结果如下：



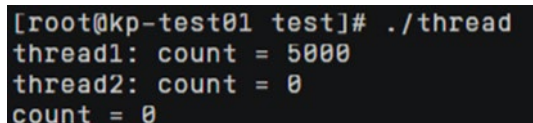
```
[root@kp-test01 test]# ./thread
thread1: count = 4748
thread2: count = 74
count = 74
```

```
[root@kp-test01 test]# ./thread
thread1: count = 3699
thread2: count = -382
count = -382
```

图 1-8 线程未加锁

结果分析：两个线程竞争访问共享的全局变量，导致每次运行的输出结果不一致。

加锁后，程序运行结果如下：



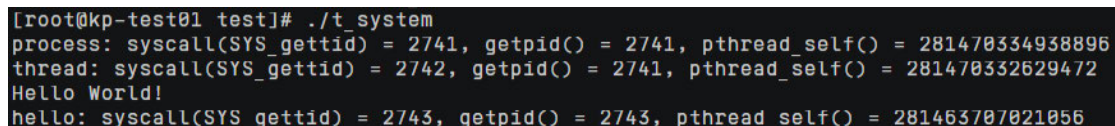
```
[root@kp-test01 test]# ./thread
thread1: count = 5000
thread2: count = 0
count = 0
```

图 1-9 线程加锁

结果分析：通过在两个线程的临界区增加互斥锁，实现了线程的同步与互斥，使得多次运行的结果都相同。

2) system 函数与 exec 族函数

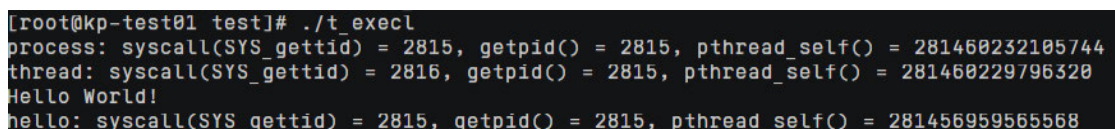
在线程中调用 system 函数后，运行结果如下：



```
[root@kp-test01 test]# ./t_system
process: syscall(SYS_gettid) = 2741, getpid() = 2741, pthread_self() = 281470334938896
thread: syscall(SYS_gettid) = 2742, getpid() = 2741, pthread_self() = 281470332629472
Hello World!
hello: syscall(SYS_gettid) = 2743, getpid() = 2743, pthread_self() = 281463707021056
```

图 1-10 线程中调用 system 函数

在线程中调用 execl 函数后，运行结果如下：



```
[root@kp-test01 test]# ./t_execl
process: syscall(SYS_gettid) = 2815, getpid() = 2815, pthread_self() = 281460232105744
thread: syscall(SYS_gettid) = 2816, getpid() = 2815, pthread_self() = 281460229796320
Hello World!
hello: syscall(SYS_gettid) = 2815, getpid() = 2815, pthread_self() = 281456959565568
```

图 1-11 线程中调用 execl 函数

结果分析：

syscall(SYS_gettid)获得的线程 tid 系统内唯一，除主线程和自己的进程一样，其它子线程都是唯一的；getpid()得到的进程 pid 系统内唯一，除了和自己的主线程一样；pthread_self()获取的线程 tid 仅在同一进程中保证唯一，有可能两个进程中都有同样一个 tid。

process 和 thread 的 pid 相同而 tid 不同，说明二者属于同一进程但不同线程；在线程中调用 system 函数后，hello 程序的 pid 与 tid 均与原来不同，说明 system 函数创建了新的进程；在线程中调用 execl 函数后，hello 程序的 pid 与 tid 均与原来相同，说明原进程的用户空间代码和数据完全被新程序替换，这与进程实验中的分析结果相同。

1.7 实验总结

1.7.1 实验中的问题与解决过程

1) 将操作系统原理课程教材 P103 习题 3.7 (图 1-1) 中的代码进行编译后报错:

```
[root@kp-test01 test]# gcc -o pid pid.c
pid.c: in function 'main':
pid.c:27:10: warning: implicit declaration of function 'wait'; did you mean 'main'? [-Wimplicit-function-declaration]
    wait(NULL);
    ^~~~~
    main
```

图 1-12 编译图 1-1 程序出错

解决方法: 发现是 wait()函数未声明, 添加头文件<wait.h>后成功通过编译。

2) 使用 Pthreads API 时编译报错, 报错信息显示 pthread 库中的函数是未定义的引用:

```
[root@kp-test01 test]# gcc -o thread thread.c
/usr/bin/ld: /tmp/cc0TtD0s.o: in function 'main':
thread.c:(.text+0x20): undefined reference to 'pthread_create'
/usr/bin/ld: thread.c:(.text+0x60): undefined reference to 'pthread_create'
/usr/bin/ld: thread.c:(.text+0x90): undefined reference to 'pthread_join'
/usr/bin/ld: thread.c:(.text+0x9c): undefined reference to 'pthread_join'
collect2: error: ld returned 1 exit status
```

图 1-13 使用 pthread 库时编译出错

解决方法: 由于 pthread 库不是 Linux 系统默认的库, 链接时需要使用库 libpthread.a, 所以使用 pthread_create 创建线程时, 要在编译中加-lpthread 参数。

例如: gcc -o thread -lpthread thread.c

参考资料: <https://blog.csdn.net/besfanfei/article/details/7542396>

3) exec 族函数的使用方法:

参考资料: <https://blog.csdn.net/u014530704/article/details/73848573>

4) 如何获取线程 tid:

解决方法: 使用 syscall(SYS_gettid)

参考资料: https://blog.csdn.net/qg_34489443/article/details/100585685

1.7.2 实验收获

通过本次实验, 我掌握了 Linux 下命令行的一些基本操作, 并熟悉了一些简单的系统调用函数, 以及 Pthreads API 的使用。此外, 我还了解了进程与线程的一些相关知识, 特别是父子进程间的继承关系、多个线程间的同步与互斥问题等。

1.7.3 意见与建议

建议老师增加实验答疑环节, 如: 通过线上腾讯会议或者线下组织集中答疑等形式。

1.8 附件

1.8.1 附件 1 程序

(一) 进程实验

1) wait 函数

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <wait.h>

int main() {
    pid_t pid, pid1;

    pid = fork();          // fork a child process

    // error occurred
    if (pid < 0) {
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    // child process
    else if (pid == 0) {
        pid1 = getpid();
        printf("child: pid = %d", pid);
        printf("child: pid1 = %d", pid1);
    }
    // parent process
    else {
        pid1 = getpid();
        printf("parent: pid = %d", pid);
        printf("parent: pid1 = %d", pid1);
        wait(NULL);
    }

    return 0;
}
```

2) 全局变量

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <wait.h>
int count = 0;

int main() {
    pid_t pid = fork();    // fork a child process

    // error occurred
    if (pid < 0) {
        fprintf(stderr, "Fork Failed\n");
        return 1;
    }
}
```

```
// child process
else if (pid == 0) {
    count--;
    printf("child: count = %d\n", count);
    printf("child: count address = %p\n", &count);
}
// parent process
else {
    wait(NULL);
    count++;
    printf("parent: count = %d\n", count);
    printf("parent: count address = %p\n", &count);
}

// before return
count++;
printf("before return: count = %d\n", count);
printf("before return: count address = %p\n", &count);

return 0;
}
```

3) system 函数与 exec 族函数

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <wait.h>
#include <stdlib.h>

int main() {
    // fork a child process
    pid_t pid = fork();

    // error occurred
    if (pid < 0) {
        fprintf(stderr, "Fork Failed\n");
        return 1;
    }
    // child process
    else if (pid == 0) {
        printf("child: pid = %d\n", getpid());
        //system("./hello");
        execl("./hello", "hello", NULL);
    }
    // parent process
    else {
        wait(NULL);
        printf("parent: pid = %d\n", getpid());
    }

    return 0;
}
```

(二) 线程实验

1) 同步与互斥

```
#include <stdio.h>
#include <pthread.h>
int count = 0;
pthread_mutex_t mutex;
void *add(void *param);
void *sub(void *param);

int main() {
    int info;
    pthread_t tid1, tid2;

    // create the mutex lock
    pthread_mutex_init(&mutex, NULL);

    // create the thread
    info = pthread_create(&tid1, NULL, add, NULL);
    if (info != 0) {
        printf("Thread1 create failed!\n");
        return -1;
    }
    info = pthread_create(&tid2, NULL, sub, NULL);
    if (info != 0) {
        printf("Thread2 create failed!\n");
        return -1;
    }

    // wait for the thread to exit
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    printf("count = %d\n", count);
    return 0;
}

void *add(void *param) {
    pthread_mutex_lock(&mutex);

    // critical section
    for (int i = 0; i < 5000; i++) count++;
    printf("thread1: count = %d\n", count);

    pthread_mutex_unlock(&mutex);
}

void *sub(void *param) {
    pthread_mutex_lock(&mutex);

    // critical section
    for (int i = 0; i < 5000; i++) count--;
    printf("thread2: count = %d\n", count);

    pthread_mutex_unlock(&mutex);
}
```

2) system 函数与 exec 族函数

```

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/syscall.h>

void *runner(void *param);

int main() {
    int info;
    pthread_t tid;

    printf("process: syscall(SYS_gettid) = %ld, getpid() = %d, pthread_self() = %ld\n",
        syscall(SYS_gettid), getpid(), pthread_self());

    // create the thread
    info = pthread_create(&tid, NULL, runner, NULL);
    if (info != 0) {
        printf("Thread create failed!\n");
        return -1;
    }

    // wait for the thread to exit
    pthread_join(tid, NULL);

    return 0;
}

void *runner(void *param) {
    printf("thread: syscall(SYS_gettid) = %ld, getpid() = %d, pthread_self() = %ld\n",
        syscall(SYS_gettid), getpid(), pthread_self());

    //system("./hello");
    execl("./hello", "hello", NULL);
}

//system或exec族函数调用的程序，可执行文件名为hello
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/syscall.h>

int main() {
    printf("Hello World!\n");

    //printf("hello: pid = %d\n", getpid()); //进程实验中使用
    printf("hello: syscall(SYS_gettid) = %ld, getpid() = %d, pthread_self() = %ld\n",
        syscall(SYS_gettid), getpid(), pthread_self()); //线程实验中使用

    return 0;
}

```

1.8.2 附件 2 Readme

一、进程实验

1) wait 函数

将操作系统原理课程教材 P103 习题 3.7 中的代码进行编译后报错：

```
[root@kp-test01 test]# gcc -o pid pid.c
pid.c: In function 'main':
pid.c:27:10: warning: implicit declaration of function 'wait'; did you mean 'main'? [-Wimplicit-function-declaration]
    wait(NULL);
    ^~~~~
    main
```

发现是 wait 函数未声明，添加头文件<wait.h>后成功通过编译。

程序运行结果如下：

```
[root@kp-test01 test]# ./pid
child: pid = 0child: pid1 = 2391parent: pid = 2391parent: pid1 = 2390[root@kp-test01 test]# ./pid
child: pid = 0child: pid1 = 2417parent: pid = 2417parent: pid1 = 2416[root@kp-test01 test]# ./pid
child: pid = 0child: pid1 = 2431parent: pid = 2431parent: pid1 = 2430[root@kp-test01 test]# ./pid
child: pid = 0child: pid1 = 2451parent: pid = 2451parent: pid1 = 2450[root@kp-test01 test]# ./pid
child: pid = 0child: pid1 = 2465parent: pid = 2465parent: pid1 = 2464[root@kp-test01 test]# ./pid
child: pid = 0child: pid1 = 2479parent: pid = 2479parent: pid1 = 2478[root@kp-test01 test]# ./pid
child: pid = 0child: pid1 = 2493parent: pid = 2493parent: pid1 = 2492[root@kp-test01 test]# ./pid
child: pid = 0child: pid1 = 2507parent: pid = 2507parent: pid1 = 2506[root@kp-test01 test]# ./pid
```

修改程序，去掉 wait 函数后，程序重新编译运行的结果如下：

```
[root@kp-test01 test]# ./pid
parent: pid = 2570parent: pid1 = 2569child: pid = 0child: pid1 = 2570[root@kp-test01 test]# ./pid
parent: pid = 2590parent: pid1 = 2589child: pid = 0child: pid1 = 2590[root@kp-test01 test]# ./pid
child: pid = 0child: pid1 = 2610parent: pid = 2610parent: pid1 = 2609[root@kp-test01 test]# ./pid
parent: pid = 2624parent: pid1 = 2623child: pid = 0child: pid1 = 2624[root@kp-test01 test]# ./pid
parent: pid = 2638parent: pid1 = 2637child: pid = 0child: pid1 = 2638[root@kp-test01 test]# ./pid
parent: pid = 2652parent: pid1 = 2651child: pid = 0child: pid1 = 2652[root@kp-test01 test]# ./pid
parent: pid = 2666parent: pid1 = 2665child: pid = 0child: pid1 = 2666[root@kp-test01 test]# ./pid
parent: pid = 2680parent: pid1 = 2679child: pid = 0child: pid1 = 2680[root@kp-test01 test]# ./pid
```

观察发现：子进程的 pid 总是为 0，且子进程的 pid1 与父进程的 pid 始终相等，父进程的 pid 始终比其 pid1 大 1。这与 fork 函数返回值的特点相符。

同时可以发现：原本的程序输出都是子进程先于父进程，但在去掉 wait 函数后，父、子进程的输出先后次序就不一定了。问题的关键在于 printf 函数的输出缓冲区：只有在缓冲区满、出现换行符、进程结束等情况下，printf 函数才会刷新缓冲区，将缓冲区中的内容显示在屏幕上。而在原本的程序中，父进程调用 wait 函数阻塞自己以等待子进程结束，故子进程会先于父进程结束，自然子进程就会比父进程先输出。去掉 wait 函数后，父、子进程哪个先结束取决于 CPU 调度，因而两者的输出次序并不一定。

相关函数与进程的概念：

- ① fork 函数：在父进程中，fork 返回新建子进程的进程 ID；在子进程中，fork 返回 0；如果出现错误，fork 返回 -1；
- ② getpid 函数：返回当前进程 ID；
- ③ wait 函数：父进程一旦调用了 wait 就立即阻塞自己，由 wait 自动分析当前进程的某个子进程是否已经退出，如果找到了这样的子进程，wait 就会收集该子进程信息，并把它彻

底销毁后返回;如果未找到这样的子进程, wait 就会一直阻塞在这里,直到有一个出现为止;

④ 僵尸进程: 一个进程使用 fork 创建子进程, 如果子进程退出, 而父进程并没有调用 wait 或 waitpid 获取子进程的状态信息, 那么子进程的进程描述符仍然保存在系统中。这种进程称之为僵尸进程;

⑤ 孤儿进程: 一个父进程退出, 而它的一个或多个子进程还在运行, 那么这些子进程将成为孤儿进程。孤儿进程将被 init 进程(进程号为 1)所收养, 并由 init 进程对它们完成状态收集工作。

2) 全局变量

初始化全局变量值为 0, 在父进程中对其执行递增操作, 在子进程中对其执行递减操作。

在父、子进程中分别输出全局变量的值和地址, 程序运行结果如下:

```
[root@kp-test01 test]# ./global
child: count = -1
child: count address = 0x420054
parent: count = 1
parent: count address = 0x420054
```

观察发现: 子进程中 count 值为-1, 父进程中 count 值为 1, 但两者地址相同。

分析: 调用 fork 函数后, 子进程完全复制父进程的地址空间, 也复制了页表, 但没有复制物理页面, 所以这时虚拟地址相同, 物理地址也相同, 但是会把父、子进程共享的页面标记为“只读”。只有当其中任何一个进程要对共享的页面执行“写操作”时, 内核才会复制一个物理页面给这个进程使用, 同时修改页表, 并把原来的只读页面标记为“可写”, 留给另外一个进程使用。这就是所谓的“写时复制”技术。

因此, 本实验中父、子进程的 count 变量虽然虚拟地址相同, 但实际的物理地址不同, 因而才有了两个不同的值。

在 return 前递增全局变量后, 输出全局变量的值和地址, 程序运行结果如下:

```
[root@kp-test01 test]# ./global
child: count = -1
child: count address = 0x420054
before return: count = 0
before return: count address = 0x420054
parent: count = 1
parent: count address = 0x420054
before return: count = 2
before return: count address = 0x420054
```

观察发现: 子进程中 count 值为 0, 父进程中 count 值为 2, 且两者地址相同。

分析: 同样是由于父、子进程中全局变量的虚拟地址相同但物理地址不同, 因此子进程中原本为-1 的 count 递增为 0, 而父进程中原本为 1 的 count 递增为 2。

3) system 函数与 exec 族函数

exec 族函数的参数:

头文件: #include <unistd.h>

- ① int execl(const char *path, const char *arg, ...);
- ② int execlp(const char *file, const char *arg, ...);
- ③ int execl_e(const char *path, const char *arg, ..., char * const envp[]);
- ④ int execl_v(const char *path, char *const argv[]);
- ⑤ int execl_vp(const char *file, char *const argv[]);
- ⑥ int execl_vpe(const char *file, char *const argv[], char *const envp[]);

参考资料: <https://blog.csdn.net/u014530704/article/details/73848573>

system 函数的参数:

头文件: #include <stdlib.h>

int system(const char * string);

调用 system 函数的运行结果如下:

```
[root@kp-test01 test]# ./p_system
child: pid = 2433
Hello World!
hello: pid = 2434
parent: pid = 2432
```

调用 execl 函数的运行结果如下:

```
[root@kp-test01 test]# ./p_execl
child: pid = 2508
Hello World!
hello: pid = 2508
parent: pid = 2507
```

观察发现: 调用 system 后 hello 程序中的 pid 比子进程 pid 大 1, 而调用 execl 后 hello 程序中的 pid 与子进程相同。

总结分析:

- ① system 函数会调用 fork 函数产生子进程, 并由子进程来执行参数字符串所代表的命令, 此命令执行完后随即返回原调用的进程;
- ② 当进程调用一种 exec 函数时, 该进程的用户空间代码和数据完全被新程序替换, 从新程序的启动例程开始执行 (当前进程剩下的程序代码便不再执行)。调用 exec 并不创建新进程, 所以调用 exec 前后该进程的 ID 并未改变。当函数调用失败时, 返回值为 -1, 调用成功则无返回值。

二、线程实验

1) 同步与互斥

将全局变量初始化为 0, 创建两个线程, 分别对全局变量进行 5000 次递增和递减操作, 并输出全局变量的值。

编译出错:

```
[root@kp-test01 test]# gcc -o thread thread.c
/usr/bin/ld: /tmp/cc0ItD0s.o: in function `main':
thread.c:(.text+0x20): undefined reference to `pthread_create'
/usr/bin/ld: thread.c:(.text+0x60): undefined reference to `pthread_create'
/usr/bin/ld: thread.c:(.text+0x90): undefined reference to `pthread_join'
/usr/bin/ld: thread.c:(.text+0x9c): undefined reference to `pthread_join'
collect2: error: ld returned 1 exit status
```

报错信息显示 pthread 库中的函数是未定义的引用。

解决方法: 由于 pthread 库不是 Linux 系统默认的库, 链接时需要使用库 libpthread.a, 所以要在编译中加-lpthread 参数。即, gcc -o thread -lpthread thread.c

参考资料: <https://blog.csdn.net/besfanfei/article/details/7542396>

正确编译后, 程序多次运行结果如下:

```
[root@kp-test01 test]# ./thread
thread1: count = 4748
thread2: count = 74
count = 74
[root@kp-test01 test]# ./thread
thread1: count = 3699
thread2: count = -382
count = -382
```

观察发现: 多次运行的输出结果不同。

分析: 两个线程竞争访问共享的全局变量, 导致每次运行的输出结果不一致。

解决方案: 通过互斥锁 mutex 来控制线程的互斥与同步。

修改代码后, 运行结果如下:

```
[root@kp-test01 test]# ./thread
thread1: count = 5000
thread2: count = 0
count = 0
```

2) system 函数与 exec 族函数

线程 tid 的获取方法: 使用 syscall(SYS_gettid)。

参考资料: https://blog.csdn.net/qg_34489443/article/details/100585685

在线程中调用 system 函数后, 运行结果如下:

```
[root@kp-test01 test]# ./t_system
process: syscall(SYS_gettid) = 2741, getpid() = 2741, pthread_self() = 281470334938896
thread: syscall(SYS_gettid) = 2742, getpid() = 2741, pthread_self() = 281470332629472
Hello World!
hello: syscall(SYS_gettid) = 2743, getpid() = 2743, pthread_self() = 281463707021056
```

在线程中调用 `execl` 函数后，运行结果如下：

```
[root@kp-test01 test]# ./t_execl
process: syscall(SYS_gettid) = 2815, getpid() = 2815, pthread_self() = 281460232105744
thread: syscall(SYS_gettid) = 2816, getpid() = 2815, pthread_self() = 281460229796320
Hello World!
hello: syscall(SYS_gettid) = 2815, getpid() = 2815, pthread_self() = 281456959565568
```

结果分析：

`syscall(SYS_gettid)`获得的线程 `tid` 系统内唯一，除主线程和自己的进程一样，其它子线程都是唯一的；`getpid()`得到的进程 `pid` 系统内唯一，除了和自己的主线程一样；`pthread_self()`获取的线程 `tid` 仅在同一进程中保证唯一，有可能两个进程中都有同样一个 `tid`。

`process` 和 `thread` 的 `pid` 相同而 `tid` 不同，说明二者属于同一进程但不同线程；在线程中调用 `system` 函数后，`hello` 程序的 `pid` 与 `tid` 均与原来不同，说明 `system` 函数创建了新的进程；在线程中调用 `execl` 函数后，`hello` 程序的 `pid` 与 `tid` 均与原来相同，说明原进程的用户空间代码和数据完全被新程序替换，这与进程实验中的分析结果相同。

2 进程通信与内存管理

2.1 实验目的

- 1) 编程实现进程的创建和软中断通信，通过观察、分析实验现象，深入理解进程及进程在调度执行和内存空间等方面的特点，掌握在 POSIX 规范中系统调用的功能和使用；
- 2) 编程实现进程的管道通信，通过观察、分析实验现象，深入理解进程管道通信的特点，掌握管道通信的同步和互斥机制；
- 3) 通过模拟实现页面置换的 FIFO 算法和 LRU 算法，深入理解两种置换算法的原理，并比较两者的优劣。

2.2 实验内容

(一) 软中断通信

使用系统调用 `fork()` 创建两个子进程，再用系统调用 `signal()` 让父进程捕捉键盘上发出的中断信号（即按 `delete/quit` 键），当父进程接收到这两个软中断的某一个后，父进程用系统调用 `kill()` 向两个子进程分别发出整数值为 16 和 17 的软中断信号，子进程获得对应软中断信号，然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 !!

父进程调用 `wait()` 函数等待两个子进程终止后，输出以下信息，结束进程执行：

Parent process is killed!!

多运行几次编写的程序，简略分析出现不同结果的原因。

(二) 管道通信

创建两个子进程，子进程 1 分 2000 次向管道写入字符 '1'，子进程 2 分 2000 次向管道写入字符 '2'。父进程等待两个子进程结束后，从管道中读出 4000 个字符并输出。

(三) 页面置换

模拟实现页面置换的 FIFO 算法和 LRU 算法，并针对某一引用串计算各自的命中率。

2.3 实验思想

通过软中断和管道两种方法，深入理解进程间是如何进行通信的。

通过模拟实现 FIFO 算法和 LRU 算法，深入理解操作系统页面置换的原理。

2.4 实验步骤

（一）软中断通信

- 1) 根据流程图编写程序，猜想一下这个程序的运行结果，然后多次运行，观察 delete/quit 键前后，会出现什么结果？分析原因。
- 2) 如果程序运行界面上显示“Child process 1 is killed by parent !! Child process 2 is killed by parent !!”，五秒之后显示“Parent process is killed !!”，怎样修改程序使得只有接收到相应的中断信号后再发生跳转，执行输出？
- 3) 将本实验中通信产生的中断通过 14 号信号值进行闹钟中断，将 signal(3,stop)当中数字信号变为 2, 体会不同中断的执行样式, 从而对软中断机制有一个更好的理解。

（二）管道通信

- 1) 先猜想 PPT 中程序的运行结果。分析管道通信是怎样实现同步与互斥的；
- 2) 然后按照 PPT 注释里的要求把代码补充完整，运行程序；
- 3) 修改程序并运行, 体会互斥锁的作用, 比较有锁和无锁程序的运行结果, 并解释之。

（三）页面置换

- 1) 模拟 FIFO 置换算法；
- 2) 模拟 LRU 置换算法；
- 3) 观察 FIFO 算法中的 Belady 异常。

2.5 测试数据设计

页面置换算法中的引用串是随机生成的。

FIFO 置换算法的 Belady 异常现象：

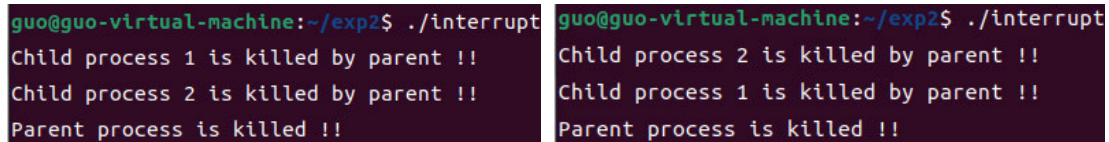
对于引用串：1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

3 帧的缺页错误数为 9，而 4 帧的缺页错误数为 10。

2.6 程序运行初值及运行结果分析

(一) 软中断通信

五秒后父进程产生时钟中断：



```
guo@guo-virtual-machine:~/exp2$ ./interrupt
Child process 1 is killed by parent !!
Child process 2 is killed by parent !!
Parent process is killed !!

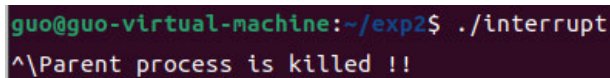
guo@guo-virtual-machine:~/exp2$ ./interrupt
Child process 2 is killed by parent !!
Child process 1 is killed by parent !!
Parent process is killed !!
```

图 2-1 五秒后时钟中断

结果分析：父进程在五秒后产生时钟中断，执行相应的 SIGALRM 信号处理函数，即：向子进程 1、2 分别发出 16、17 号信号。然后，子进程在收到相应信号后从挂起状态恢复，并执行自定义的信号处理函数，即：输出各自的语句（哪个子进程先输出取决于 CPU 调度）。最后，父进程在两个子进程结束后再进行输出。

注：使用 pause 函数将子进程挂起，可以让子进程只有接收到相应的中断信号后再发生跳转并执行输出。

五秒内键盘发出 SIGQUIT 信号（信号处理函数与 SIGALRM 信号相同）：



```
guo@guo-virtual-machine:~/exp2$ ./interrupt
^\\Parent process is killed !!
```

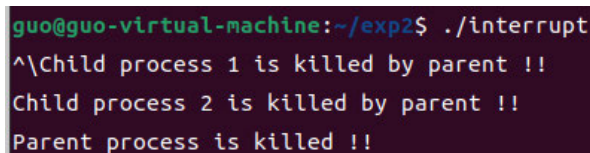
图 2-2 五秒内键盘发出 SIGQUIT 信号但只有父进程输出

观察发现：只有父进程输出，两个子进程均无输出，但能够正常结束。

结果分析：当从键盘上发送 SIGQUIT 信号时，子进程也收到了该信号，但由于子进程并未设置相应的信号处理函数，因此执行默认的结束进程操作。

解决方法：在子进程中添加 signal(SIGQUIT, SIG_IGN) 语句，屏蔽 SIGQUIT 信号。

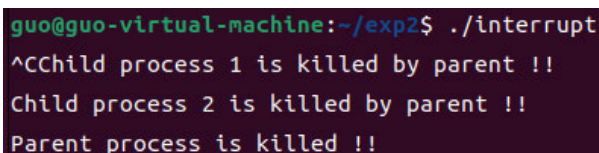
程序修改后的运行结果如下：



```
guo@guo-virtual-machine:~/exp2$ ./interrupt
^\\Child process 1 is killed by parent !!
Child process 2 is killed by parent !!
Parent process is killed !!
```

图 2-3 五秒内键盘发出 SIGQUIT 信号且正常输出

使用 SIGINT 信号代替 SIGQUIT 信号进行中断的程序运行结果如下：



```
guo@guo-virtual-machine:~/exp2$ ./interrupt
^CChild process 1 is killed by parent !!
Child process 2 is killed by parent !!
Parent process is killed !!
```

图 2-4 五秒内键盘发出 SIGINT 信号

(二) 管道通信

管道未加锁时，程序运行结果如下（截取部分）：

121
212
121
222

图 2-5 管道未加锁

观察发现：两个子进程交替向管道输出字符。

管道加锁后，程序运行结果如下（截取部分）：

111111111111111111111111111111111111	222222222222222222222222222222222222
111111111111111111111111111111111111	222222222222222222222222222222222222
222222222222222222222222222222222222	111111111111111111111111111111111111
222222222222222222222222222222222222	111111111111111111111111111111111111

a) 子进程 1 先写入

b) 子进程 2 先写入

图 2-6 管道加锁

观察发现：都是先输出 2000 个相同字符后，再输出 2000 个另一种相同字符。

结果分析：哪个子进程先输出取决于 CPU 的进程调度，但由于使用了 `lockf()` 函数对管道加锁，因此子进程的字符输出都是连续的。

(三) 页面置换

1) FIFO 置换算法

```
reference:
3 6 7 5 3 5 6 2 9 1 2 7 0 9 3 6 0 6 2 6

FIFO:

Page 3 arrives but not in the frame.
frame: |3|

Page 6 arrives but not in the frame.
frame: |3| |6|

Page 7 arrives but not in the frame.
frame: |3| |6| |7|

Page 5 arrives but not in the frame.
frame: |3| |6| |7| |5|

Page 3 arrives and in the frame.
Page 5 arrives and in the frame.
Page 6 arrives and in the frame.
Page 2 arrives but not in the frame.
frame: |6| |7| |5| |2|

Page 9 arrives but not in the frame.
frame: |7| |5| |2| |9|

Page 1 arrives but not in the frame.
frame: |5| |2| |9| |1|

Page 2 arrives and in the frame.
Page 7 arrives but not in the frame.
frame: |2| |9| |1| |7|

Page 0 arrives but not in the frame.
frame: |9| |1| |7| |0|

Page 9 arrives and in the frame.
Page 3 arrives but not in the frame.
frame: |1| |7| |0| |3|

Page 6 arrives but not in the frame.
frame: |7| |0| |3| |6|

Page 0 arrives and in the frame.
Page 6 arrives and in the frame.
Page 2 arrives but not in the frame.
frame: |0| |3| |6| |2|

Page 6 arrives and in the frame.

Page fault: 12
ratio: 40.00%
```

图 2-7 FIFO 置换算法实例

缺页错误数为 12, 命中率为 40.00%

FIFO 置换算法的流程图如下：

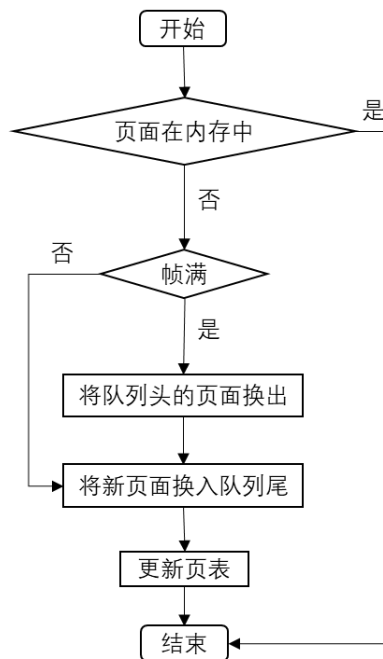


图 2-8 FIFO 算法流程图

FIFO 算法创建一个队列来管理所有的内存页面，置换的是队列的首个页面。当需要调入页面到内存时，就将其加到队列的尾部。

2) LRU 置换算法

```

reference:
3 6 7 5 3 5 6 2 9 1 2 7 0 9 3 6 0 6 2 6

LRU:

Page 3 arrives but not in the frame.
frame: [3]
Page 6 arrives but not in the frame.
frame: [3] [6]
Page 7 arrives but not in the frame.
frame: [3] [6] [7]
Page 5 arrives but not in the frame.
frame: [3] [6] [7] [5]
Page 3 arrives and in the frame.
Page 5 arrives and in the frame.
Page 6 arrives and in the frame.
Page 2 arrives but not in the frame.
frame: [3] [6] [2] [5]
Page 9 arrives but not in the frame.
frame: [9] [6] [2] [5]
Page 1 arrives but not in the frame.

```

```

frame: [9] [6] [2] [1]
Page 2 arrives and in the frame.
Page 7 arrives but not in the frame.
frame: [9] [7] [2] [1]
Page 0 arrives but not in the frame.
frame: [0] [7] [2] [1]
Page 9 arrives but not in the frame.
frame: [0] [7] [2] [9]
Page 3 arrives but not in the frame.
frame: [0] [7] [3] [9]
Page 6 arrives but not in the frame.
frame: [0] [6] [3] [9]
Page 0 arrives and in the frame.
Page 6 arrives and in the frame.
Page 2 arrives but not in the frame.
frame: [0] [6] [3] [2]
Page 6 arrives and in the frame.

Page fault: 13
ratio: 35.00%

```

图 2-9 LRU 置换算法实例

缺页错误数为 13，命中率为 35.00%

LRU 置换算法（计数器实现）的流程图如下：

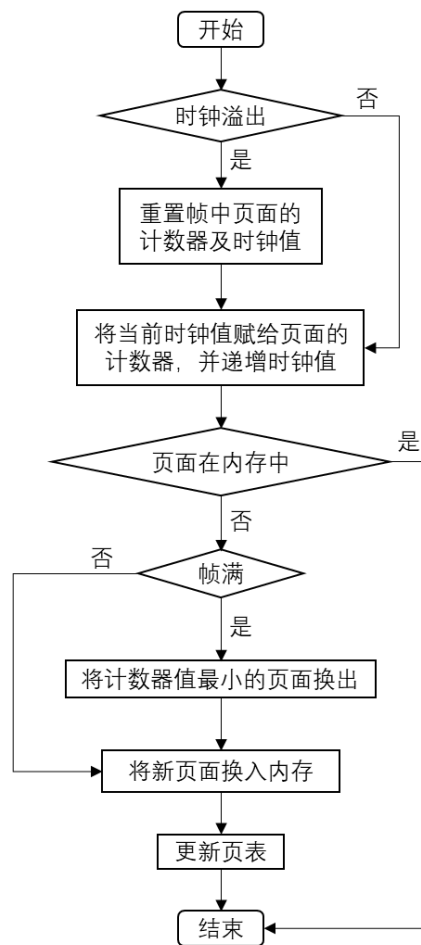


图 2-10 LRU 算法（计数器实现）流程图

LRU 置换将每个页面与它的上次使用的时间关联起来。当需要置换页面时，LRU 选择最长时间没有使用的页面。LRU 算法的一个重要问题是如何确定由上次使用时间定义的帧的顺序，有两个实现是可行的。第一种方案是堆栈方法，即：每当页面被引用时，它就从堆栈中移除并放在顶部。第二种方案是计数器方法，即：为每个页表条目关联一个使用时间域，并为 CPU 添加一个逻辑时钟或计数器，每次内存引用都会递增时钟，并将时钟寄存器的内容复制到对应页表条目的使用时间域。

需要注意的是，当选择计数器实现时，可能会存在时钟溢出的问题。这里的解决方案是，当时钟达到最大值时，将帧中的页面按照各自的计数器大小升序排序，并依次重置计数器为 0、1、2、3……同时，将时钟值重置为帧中的页面个数。

FIFO 置换与 LRU 置换相比，前者易于理解和编程，但性能并不总是十分理想，FIFO 置换的页面可能包含一个被大量使用的、早已初始化的变量，但仍在不断使用；而 LRU 置换则需要由硬件来辅助完成，否则如果采用中断来更新时钟域或堆栈都开销很大。

3) Belady 异常

```
reference:
1 2 3 4 1 2 5 1 2 3 4 5

FIFO(Belady):

Page 1 arrives but not in the frame.
frame: |1|
Page 2 arrives but not in the frame.
frame: |1| |2|
Page 3 arrives but not in the frame.
frame: |1| |2| |3|
Page 4 arrives but not in the frame.
frame: |2| |3| |4|
Page 1 arrives but not in the frame.
frame: |3| |4| |1|
Page 2 arrives but not in the frame.
frame: |4| |1| |2|
Page 5 arrives but not in the frame.
frame: |1| |2| |5|
Page 1 arrives and in the frame.
Page 2 arrives and in the frame.
Page 3 arrives but not in the frame.
frame: |2| |5| |3|
Page 4 arrives but not in the frame.
frame: |5| |3| |4|
Page 5 arrives and in the frame.

Page fault: 9
ratio: 25.00%
```

a) 3 帧

```
reference:
1 2 3 4 1 2 5 1 2 3 4 5

FIFO(Belady):

Page 1 arrives but not in the frame.
frame: |1|
Page 2 arrives but not in the frame.
frame: |1| |2|
Page 3 arrives but not in the frame.
frame: |1| |2| |3|
Page 4 arrives but not in the frame.
frame: |1| |2| |3| |4|
Page 1 arrives and in the frame.
Page 2 arrives and in the frame.
Page 5 arrives but not in the frame.
frame: |2| |3| |4| |5|
Page 1 arrives but not in the frame.
frame: |3| |4| |5| |1|
Page 2 arrives but not in the frame.
frame: |4| |5| |1| |2|
Page 3 arrives but not in the frame.
frame: |5| |1| |2| |3|
Page 4 arrives but not in the frame.
frame: |1| |2| |3| |4|
Page 5 arrives but not in the frame.
frame: |2| |3| |4| |5|

Page fault: 10
ratio: 16.67%
```

b) 4 帧

图 2-11 FIFO 置换算法的 Belady 异常

所谓 Belady 异常指的是，对于有些页面置换算法（如 FIFO），随着分配帧数量的增加，缺页错误率可能会增加。

如图 2-11 所示，3 帧时，缺页错误数为 9；而 4 帧时，缺页错误数为 10。

2.7 页面置换算法复杂度分析

假设内存中总共有 n 帧。

1) FIFO 置换算法

FIFO 置换算法使用队列来实现，当发生缺页错误时，将新换入的页面放在队列尾，并置换出队列头的页面，因此该算法的时间复杂度为 $O(1)$ 。使用数组实现循环队列时，为了区分队列的空与满，队列的长度应比帧数大 1，因此空间复杂度为 $O(n+1)$ 。

2) LRU 置换算法

LRU 置换算法使用计数器来实现，当发生缺页错误时，置换具有最小时间的页面，这种方案需要搜索帧中的页面以查找需要置换的 LRU 页面。因此该算法的时间复杂度为 $O(n)$ ，空间复杂度也为 $O(n)$ 。

2.8 回答问题

2.8.1 软中断通信

1) 你最初认为运行结果会怎么样？写出你猜测的结果；

5 秒后中断是由父进程的时钟信号产生的，相应的信号处理函数向两个子进程发出 16、17 号信号，同时父进程调用 wait 函数阻塞自己，因此子进程先输出“Child process 1/2 is killed by parent !!”，父进程后输出“Parent process is killed!!”。5 秒内中断是由键盘发出 SIGQUIT 信号产生的，由于信号处理函数与 SIGALRM 信号相同，因此运行结果应该与 5 秒后中断相同。

2) 实际的结果什么样？有什么特点？在接收不同中断前后有什么差别？请将 5 秒内中断和 5 秒后中断的运行结果截图，试对产生该现象的原因进行分析。

5 秒后中断的运行结果与预期相符，如图 2-1 所示。但 5 秒内中断只有父进程输出，两个子进程均无输出，如图 2-2 所示。原因分析：两个子进程也收到了键盘发出的 SIGQUIT 信号，但未定义相应的信号处理函数，因此执行了默认的结束进程操作。解决方法是在两个子进程中添加 `signal(SIGQUIT, SIG_IGN)` 语句，屏蔽 SIGQUIT 信号。最终的运行结果如图 2-3 所示。

3) 针对实验过程 2，怎样修改的程序？修改前后程序的运行结果是什么？请截图说明。

修改程序前，程序运行界面上显示“Child process 1 is killed by parent !! Child process 2 is killed by parent !!”，五秒之后显示“Parent process is killed !!”。修改程序后，在子进程中使用 `pause` 函数，以挂起进程并等待信号，这样只有在收到信号后才会发生跳转，运行结果如图 2-3 所示。

4) 针对实验过程 3，程序运行的结果是什么样子？时钟中断有什么不同？

使用时钟中断进行通信时，在经过 5 秒后，父进程收到 SIGALRM 信号，然后执行相应的信号处理函数，程序运行结果如图 2-1 所示。调用 `alarm` 函数设置时钟中断并不会使进程暂停执行，而是在经过设置的时间后向调用进程发送 SIGALRM 信号，其默认的信号处理函数是终止进程。使用 SIGINT 信号替换 SIGQUIT 信号的运行结果如图 2-4 所示。

5) kill 命令在程序中使用了幾次？每次的作用是什么？执行后的现象是什么？

kill 命令在程序中使用了 2 次。第一次是向子进程 1 发送 16 号信号，第二次是向子进程 2 发送 17 号信号。子进程在收到信号后，会执行相应的信号处理函数（在本程序中是输出语句“Child process 1/2 is killed by parent !!”）。如果没有自定义信号处理函数，子进程在收到信号后会执行系统默认的信号处理函数。

6) 使用 kill 命令可以在进程的外部杀死进程。进程怎样能主动退出？这两种退出方式哪种更好一些？

进程在函数中 return 或调用 exit 函数都可以主动退出；而使用 kill 命令则是强制退出。进程主动退出比较好，因为若子在子进程退出前使用 kill 命令杀死其父进程，子进程就会变成孤儿进程，系统会让 init 进程接管子进程；当使用 kill 命令使得子进程先于父进程退出时，若父进程又没有调用 wait 函数等待子进程结束，则子进程会处于僵死状态，且会一直保持下去，直到系统重启。子进程处于僵死状态时，内核会保存该进程的一些必要信息，此时子进程始终占用着资源，同时减少了系统可以创建的最大进程数，当僵死的子进程占用资源过多时，很可能导致系统卡死。

2.8.2 管道通信

1) 你最初认为运行结果会怎么样？

预期结果：先输出 2000 个字符‘1’后，再输出 2000 个字符‘2’。

2) 实际的结果什么样？有什么特点？试对产生该现象的原因进行分析。

管道未加锁时，两个子进程交替向管道写入字符。原因：两个子进程轮流抢占 CPU，导致对管道写入操作的不连续。

管道加锁后，有时候子进程 1 先写入，子进程 2 后写入，有时候则恰恰相反。但都是先写入 2000 个相同字符后，再写入 2000 个另一种相同字符。原因：由于 CPU 的进程调度不同，两个子进程谁先写入并不一定，但是由于在子进程中使用了 lockf 函数对管道加锁，故在某一子进程的写入操作未完成时，即使另一子进程抢占了 CPU，也无法向管道中写入字符，直至前者完成写操作并对管道解锁。

3) 实验中管道通信是怎样实现同步与互斥的？如果不控制同步与互斥会发生什么后果？

实验中使用了 lockf 函数实现同步与互斥，当子进程对管道进行写操作时，对管道的写端加锁，写操作执行完成后再进行解锁。如果不控制同步与互斥，由于子进程抢占 CPU，会导致两个子进程交替地对管道的写入端进行写操作。

2.9 实验总结

2.9.1 实验中的问题与解决过程

1) 不了解相关函数（如：signal、kill、pipe 等）的使用方法。

解决方法：通过 man 指令或者上网查阅相关资料。

signal 与 kill 函数的使用方法：<https://zhuanlan.zhihu.com/p/113876980>

pipe 函数的使用方法：<https://www.cnblogs.com/kunhu/p/3608109.html>

2) 如何通过键盘向进程发送信号。

解决方法：delete 会向进程发送 SIGINT 信号，quit 会向进程发送 SIGQUIT 信号。ctrl+c 为 delete，ctrl+\ 为 quit。

参考资料：<https://blog.csdn.net/mylzh/article/details/38385739>

3) 如何控制管道的同步与互斥。

解决方法：当子进程对管道执行写操作时，使用 lockf 函数对管道的写入端加锁，直到写操作完成后再对管道解锁。

lockf 函数使用方法：<https://zhuanlan.zhihu.com/p/427506654>

4) 使用计数器实现的 LRU 算法如何解决时钟溢出问题。

解决方法：当时钟达到最大值时，将帧中的页面按照各自的计数器大小升序排序，并依次重置计数器为 0、1、2、3……同时，将时钟值重置为帧中的页面个数。

2.9.2 实验收获

通过本次实验，我进一步熟悉了 Linux 系统环境，并了解了软中断通信与管道通信的实现方法。此外，通过对 FIFO 置换算法与 LRU 置换算法的编程模拟，我对于操作系统中页面置换的原理有了更深刻的理解，并了解了这两种算法各自的优劣。

2.9.3 意见与建议

在实验过程中，我遇到的一个主要问题是难以理解实验 PPT 中的任务要求，特别是实验 PPT 流程图与实验内容不相符、部分语义表述模糊。比如，软中断通信中，实验过程 3 要求将通信产生的中断通过 14 号信号值进行闹钟中断，但这究竟是指将父进程收到的 SIGQUIT 信号换成 SIGALRM 信号，还是指父进程发送 SIGALRM 信号给子进程？因此，建议老师进一步完善实验 PPT，明确实验步骤与内容。

2.10 附件

2.10.1 附件 1 程序

(一) 软中断通信

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
#include <stdlib.h>
pid_t pid1, pid2;
void parent() {
    kill(pid1, 16);
    kill(pid2, 17);
}
void child1() {
    printf("Child process 1 is killed by parent !!\n");
    exit(0);
}
void child2() {
    printf("Child process 2 is killed by parent !!\n");
    exit(0);
}

int main() {
    while ((pid1 = fork()) == -1);
    if (pid1 > 0) {
        while ((pid2 = fork()) == -1);
        if (pid2 > 0) {
            //父进程
            signal(SIGQUIT, parent);
            //signal(SIGINT, parent);
            signal(SIGALRM, parent);
            alarm(5);
            wait(NULL);
            wait(NULL);
            printf("Parent process is killed !!\n");
            exit(0);
        } else {
            //子进程2
            signal(17, child2);
            signal(SIGQUIT, SIG_IGN);
            //signal(SIGINT, SIG_IGN);
            pause();
        }
    } else {
        //子进程1
        signal(16, child1);
        signal(SIGQUIT, SIG_IGN);
        //signal(SIGINT, SIG_IGN);
        pause();
    }
}
```

(二) 管道通信

```
#include <unistd.h>
#include <stdio.h>
#include <wait.h>
#include <stdlib.h>

int pid1, pid2;

int main() {

    int fd[2];
    char InPipe[5000];           //定义读缓冲区
    char c1 = '1', c2 = '2';
    pipe(fd);                    //创建管道

    while ((pid1 = fork()) == -1);

    if (pid1 == 0) {
        //子进程1
        lockf(fd[1], 1, 0);      //锁定管道

        for (int i = 0; i < 2000; i++)
            write(fd[1], &c1, 1);
        sleep(5);

        lockf(fd[1], 0, 0);      //解除锁定
        exit(0);
    } else {

        while ((pid2 = fork()) == -1);

        if (pid2 == 0) {
            //子进程2
            lockf(fd[1], 1, 0);    //锁定管道

            for (int i = 0; i < 2000; i++)
                write(fd[1], &c2, 1);
            sleep(5);

            lockf(fd[1], 0, 0);    //解除锁定
            exit(0);
        } else {
            //父进程
            wait(NULL);
            wait(NULL);

            read(fd[0], InPipe, 4000);
            InPipe[4000] = '\0';
            printf("%s\n", InPipe);
            exit(0);
        }
    }

    return 0;
}
```

(三) 页面置换

```

#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <time.h>
#include <limits.h>

//页表项
typedef struct page {
    unsigned int frame;      //帧号
    bool flag;              //标记位
    unsigned int counter;    //计数器
} page;

page *page_table = NULL;    //页表指针
unsigned int *reference = NULL; //页面引用串
unsigned int table_size = 10; //页表大小
unsigned int frame_size = 4;  //帧表大小
unsigned int ref_size = 20;   //引用串大小

//Belady异常的引用串实例
unsigned int belady[12] = { 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 };

//初始化页表，将标记位置无效，计数器清零
void initial_page_table(void) {
    //分配页表空间
    page_table = (page *)malloc(table_size * sizeof(page));
    if (!page_table) {
        printf("page_table: malloc failed!\n");
        exit(0);
    }

    //初始化页表项
    for (unsigned int i = 0; i < table_size; i++) {
        page_table[i].flag = false;
        page_table[i].counter = 0;
    }
}

//初始化引用串
void initial_reference(void) {
    //分配引用串空间
    reference = (unsigned int *)malloc(ref_size * sizeof(unsigned int));
    if (!reference) {
        printf("reference: malloc failed!\n");
        exit(0);
    }

    //生成随机引用串序列
    srand((unsigned int)time(0)); //设置随机数种子
    for (unsigned int i = 0; i < ref_size; i++) {
        reference[i] = rand() % table_size;
    }
}

```



```

//输出引用串
void print_reference(void) {
    printf("\nreference:\n");
    for (unsigned int i = 0; i < ref_size; i++) {
        printf("%u ", reference[i]);
    }
    printf("\n");
}

//释放页表和引用串的内存
void release(void) {
    if (page_table)
        free(page_table);
    if (reference && reference != belady)
        free(reference);
}

//设置Belady异常的引用串实例相关参数
void Belady(void) {
    ref_size = 12;
    frame_size = 3;
    table_size = 6;
    reference = belady;
}

int FIFO(void) {
    unsigned int diseffect = 0;    //缺页错误数
    unsigned int front = 0;        //队列头
    unsigned int rear = 0;         //队列尾
    unsigned int queue_size = frame_size + 1;    //队列大小

    //分配队列空间
    unsigned int *queue = (unsigned int *)malloc(queue_size * sizeof(unsigned int));
    if (!queue) {
        printf("queue: malloc failed!\n");
        exit(0);
    }

    for (unsigned int i = 0; i < ref_size; i++) {
        printf("Page %u arrives ", reference[i]);
        //页面不在帧中
        if (!page_table[reference[i]].flag) {
            printf("but not in the frame.\n");
            diseffect++;

            if ((rear + 1) % queue_size == front) {
                //帧满换出
                page_table[queue[front]].flag = false;
                front = (front + 1) % queue_size;
            }
            //页面换入
            page_table[reference[i]].flag = true;
            page_table[reference[i]].frame = rear;
            queue[rear] = reference[i];
            rear = (rear + 1) % queue_size;
        }
    }
}

```

```

        //输出当前帧中内容
        printf("frame: ");
        for (unsigned int k = front; k != rear; k = (k + 1) % queue_size)
            printf("|%u| ", queue[k]);
        printf("\n");
    } else {
        //页面在帧中
        printf("and in the frame.\n");
    }
}

free(queue);
return diseffect;
}

//查找最早到达的页面
int find_min(unsigned int *frame) {
    unsigned int min = UINT_MAX, frame_num = 0;
    for (unsigned int i = 0; i < frame_size; i++) {
        if (page_table[frame[i]].counter < min) {
            min = page_table[frame[i]].counter;
            frame_num = i;
        }
    }
    return frame_num;
}

unsigned int LRU(void) {
    unsigned int diseffect = 0;    //缺页错误数
    unsigned int clock = 0;       //逻辑时钟
    unsigned int full = 0;        //帧满标识

    unsigned int *frame = (unsigned int *)malloc(frame_size * sizeof(unsigned int));
    if (!frame) {
        printf("frame: malloc failed!\n");
        exit(0);
    }

    for (unsigned int i = 0; i < ref_size; i++) {
        printf("Page %u arrives ", reference[i]);
        //考虑时钟溢出
        if (clock == UINT_MAX) {
            unsigned int min;
            unsigned int *temp = (unsigned int *)malloc(frame_size * sizeof(unsigned
int));

            for (unsigned int i = 0; i < frame_size; i++) {
                min = find_min(frame);
                temp[i] = frame[min];
                page_table[frame[min]].counter = UINT_MAX;
            }
            for (unsigned int i = 0; i < frame_size; i++) {
                page_table[temp[i]].counter = i;
            }
            free(temp);
            clock = frame_size;
        }
    }
}

```

```

    page_table[reference[i]].counter = clock;
    clock++;
    //页面不在帧中
    if (!page_table[reference[i]].flag) {
        printf("but not in the frame.\n");
        diseffect++;
        if (full < frame_size) {
            //帧未满时
            page_table[reference[i]].flag = true;
            page_table[reference[i]].frame = full;
            frame[full] = reference[i];
            full++;
        } else {
            unsigned int replace = find_min(frame);
            //页面换出
            page_table[frame[replace]].flag = false;
            //页面换入
            page_table[reference[i]].flag = true;
            page_table[reference[i]].frame = replace;
            frame[replace] = reference[i];
        }
        //输出当前帧中内容
        printf("frame: ");
        for (unsigned int k = 0; k < full; k++)
            printf("|%u| ", frame[k]);
        printf("\n");
    } else {
        //页面在帧中
        printf("and in the frame.\n");
    }
}

free(frame);
return diseffect;
}

//设置参数
char set_parameter(void) {
    //置换算法或Belady异常
    printf("\nChoice: 1)FIFO      2)LRU      3)FIFO(Belady)\n");
    char choice = getchar();
    while (getchar() != '\n');
    if (choice == '3') {
        Belady();
        return choice;
    }
}

unsigned int size = 0;
//页表大小
printf("Please enter the size(0 < size <= 128) of the page_table: ");
if (scanf("%u", &size) == 1 && size > 0 && size <= 128)
    table_size = size;
else {
    table_size = 10;
    printf("default = 10\n");
}
while (getchar() != '\n');

```

```

//帧表大小
printf("Please enter the size(0 < size <= 32) of the frame: ");
if (scanf("%u", &size) == 1 && size > 0 && size <= 32)
    frame_size = size;
else {
    frame_size = 4;
    printf("default = 4\n");
}
while (getchar() != '\n');

//引用串大小
printf("Please enter the size(0 <= size <= 100) of the reference: ");
if (scanf("%u", &size) == 1 && size >= 0 && size <= 100)
    ref_size = size;
else {
    ref_size = 20;
    printf("default = 20\n");
}
while (getchar() != '\n');

return choice;
}

int main(void) {
    printf("*****\n");
    printf("    PAGE REPLACEMENT\n");
    printf("*****\n\n");

    char choicel = set_parameter();
    if (choicel != '3')
        initial_reference();
    unsigned int diseffect;

    while (true) {
        print_reference();
        initial_page_table();

        switch (choicel) {
            case '1':
                printf("\nFIFO:\n\n");
                diseffect = FIFO();
                break;
            case '2':
                printf("\nLRU:\n\n");
                diseffect = LRU();
                break;
            case '3':
                printf("\nFIFO(Belady):\n\n");
                diseffect = FIFO();
                break;
            default:
                printf("\ndefault = FIFO:\n\n");
                diseffect = FIFO();
        }
    }
}

```

```

//输出缺页错误数及命中率
printf("\nPage fault: %d\n", diseffect);
printf("ratio: %.2f%%\n", (1.0 - (float)diseffect / ref_size) * 100.0);

printf("\nQuit? [default[Enter] = 'No', [y] = 'yes']: ");
//退出程序
if (getchar() == 'y') {
    release();
    break;
}

printf("\nReset? [default[Enter] = 'No', [y] = 'yes']: ");
if (getchar() == 'y') {
    while (getchar() != '\n');
    //重置参数
    release();
    choicel = set_parameter();
    if (choicel != '3')
        initial_reference();
} else {
    printf("\n1)Change the algorithm      2)Change the size of frame\n");
    char choice2 = getchar();
    while (getchar() != '\n');

    //当选择更改帧表大小时，size存储从键盘输入的数
    unsigned int size = 0;

    switch (choice2) {
        case '1':
            //更改算法
            choicel = (choicel == '2') ? '1' : '2';
            break;
        case '2':
            //更改帧表大小
            printf("\nThe new size(0 < size <= 32) of the frame: ");
            if (scanf("%u", &size) == 1 && size > 0 && size <= 32)
                frame_size = size;
            else {
                frame_size = 4;
                printf("default = 4\n");
            }
            while (getchar() != '\n');
            break;
        default:
            printf("default: Change the algorithm\n");
            choicel = (choicel == '2') ? '1' : '2';
    }
}

return 0;
}

```

2.10.2 附件 2 Readme

一、软中断通信

实验前准备：学习 man 命令的用法，通过它查看 fork、kill、signal、sleep、exit 等系统调用的在线帮助，并阅读参考资料，复习 C 语言的相关内容。

```

KILL(1)                                User Commands                                KILL(1)

NAME
    kill - send a signal to a process

SYNOPSIS
    kill [options] <pid> [...]

DESCRIPTION
    The default signal for kill is TERM. Use -l or -L to list available
    signals. Particularly useful signals include HUP, INT, KILL, STOP,
    CONT, and 0. Alternate signals may be specified in three ways: -9,
    -SIGKILL or -KILL. Negative PID values may be used to choose whole
    process groups; see the PGID column in ps command output. A PID of -1
    is special; it indicates all processes except the kill process itself
    and init.

OPTIONS
    <pid> [...]
        Send signal to every <pid> listed.

    -<signal>
    -s <signal>

Manual page kill(1) line 1 (press h for help or q to quit)

```

相关函数介绍：

1) sighandler_t signal(int signum, sighandler_t handler);

头文件：#include <signal.h>

按照参数 signum 指定的信号编号来设置该信号的处理函数。当指定的信号到达时，就会跳转到参数 handler 指定的函数执行。

2) int kill(pid_t pid, int sig);

头文件：#include <signal.h>

参数 pid 可能选择有以下四种：

pid > 0 时，pid 是信号欲送往的进程的标识。

pid == 0 时，信号将送往所有与调用 kill() 的那个进程属同一个使用组的进程。

pid == -1 时，信号将送往所有调用进程有权给其发送信号的进程，除了进程 1(init)。

pid < -1 时，信号将送往以 -pid 为组标识的进程。

参数 sig 是准备发送的信号代码，假如其值为零则没有任何信号送出，但是系统会执行

错误检查，通常会利用 sig 值为零来检验某个进程是否仍在执行。

3) int pause(void);

头文件: #include <unistd.h>

挂起本进程以等待信号，接收到信号后恢复执行。当接收到中止进程信号时，该调用不再返回。

4) unsigned int alarm(unsigned int seconds);

头文件: #include <unistd.h>

alarm 函数的主要功能是设置信号传送闹钟，即用来设置信号 SIGALRM 在经过参数 seconds 秒数后发送给目前的进程。如果未设置信号 SIGALARM 的处理函数，那么 alarm 默认处理是终止进程。

遇到的问题：如何通过键盘发送信号。

解决方法：delete 会向进程发送 SIGINT 信号，quit 会向进程发送 SIGQUIT 信号。ctrl+c 为 delete，ctrl+\ 为 quit。

参考资料：<https://blog.csdn.net/mylizh/article/details/38385739>

五秒后父进程产生时钟中断的运行结果如下：

```
guo@guo-virtual-machine:~/exp2$ ./interrupt
Child process 1 is killed by parent !!
Child process 2 is killed by parent !!
Parent process is killed !!

guo@guo-virtual-machine:~/exp2$ ./interrupt
Child process 2 is killed by parent !!
Child process 1 is killed by parent !!
Parent process is killed !!
```

结果分析：父进程在五秒后产生时钟中断，执行相应的 SIGALRM 信号处理函数，即：向子进程 1、2 分别发出 16、17 号信号。然后，子进程在收到相应信号后从挂起状态恢复，并执行自定义的信号处理函数，即：输出各自的语句（哪个子进程先输出取决于 CPU 调度）。最后，父进程在两个子进程结束后再进行输出。

注：使用 pause 函数将子进程挂起，可以让子进程只有接收到相应的中断信号后再发生跳转并执行输出。

五秒内键盘发出 SIGQUIT 信号（信号处理函数与 SIGALRM 信号相同）的运行结果如下：

```
guo@guo-virtual-machine:~/exp2$ ./interrupt
^\\Parent process is killed !!
```

观察发现：只有父进程输出，两个子进程均无输出，但能够正常结束。

结果分析：当从键盘上发送 SIGQUIT 信号时，子进程也收到了该信号，但由于子进程并未设置相应的信号处理函数，因此执行默认的结束进程操作。

解决方法：在子进程中添加 `signal(SIGQUIT, SIG_IGN)` 语句，屏蔽 SIGQUIT 信号。

程序修改后的运行结果如下：

```
guo@guo-virtual-machine:~/exp2$ ./interrupt
^\\Child process 1 is killed by parent !!
Child process 2 is killed by parent !!
Parent process is killed !!
```

使用 SIGINT 信号代替 SIGQUIT 信号进行中断的程序运行结果如下：

```
guo@guo-virtual-machine:~/exp2$ ./interrupt
^CChild process 1 is killed by parent !!
Child process 2 is killed by parent !!
Parent process is killed !!
```

二、管道通信

实验前准备：学习 `man` 命令的用法，通过它查看管道创建、同步互斥系统调用的在线帮助，并阅读参考资料。

```
PIPE(2)                                Linux Programmer's Manual                                PIPE(2)

NAME
    pipe, pipe2 - create pipe

SYNOPSIS
    #include <unistd.h>

    /* On Alpha, IA-64, MIPS, SuperH, and SPARC/SPARC64; see NOTES */
    struct fd_pair {
        long fd[2];
    };
    struct fd_pair pipe();

    /* On all other architectures */
    int pipe(int pipefd[2]);

    #define _GNU_SOURCE                  /* See feature_test_macros(7) */
    #include <fcntl.h>                  /* Obtain O_* constant definitions */
    #include <unistd.h>

    int pipe2(int pipefd[2], int flags);

Manual page pipe(2) line 1/182 11% (press h for help or q to quit)
```


三、页面置换

1) FIFO 置换算法

```
reference:
3 6 7 5 3 5 6 2 9 1 2 7 0 9 3 6 0 6 2 6

FIFO:

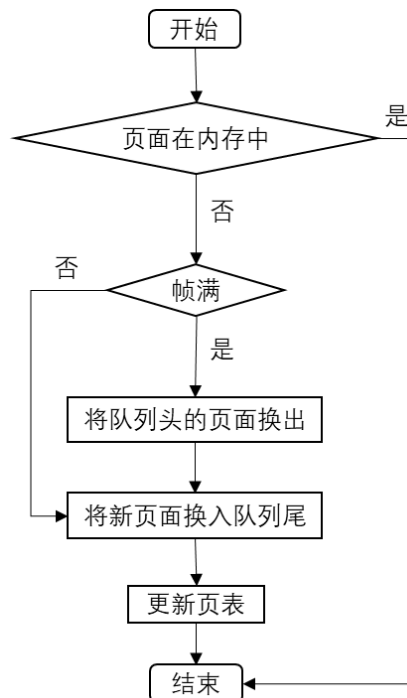
Page 3 arrives but not in the frame.
frame: [3]
Page 6 arrives but not in the frame.
frame: [3] [6]
Page 7 arrives but not in the frame.
frame: [3] [6] [7]
Page 5 arrives but not in the frame.
frame: [3] [6] [7] [5]
Page 3 arrives and in the frame.
Page 5 arrives and in the frame.
Page 6 arrives and in the frame.
Page 2 arrives but not in the frame.
frame: [6] [7] [5] [2]
Page 9 arrives but not in the frame.
frame: [7] [5] [2] [9]

Page 1 arrives but not in the frame.
frame: [5] [2] [9] [1]
Page 2 arrives and in the frame.
Page 7 arrives but not in the frame.
frame: [2] [9] [1] [7]
Page 0 arrives but not in the frame.
frame: [9] [1] [7] [0]
Page 9 arrives and in the frame.
Page 3 arrives but not in the frame.
frame: [1] [7] [0] [3]
Page 6 arrives but not in the frame.
frame: [7] [0] [3] [6]
Page 0 arrives and in the frame.
Page 6 arrives and in the frame.
Page 2 arrives but not in the frame.
frame: [0] [3] [6] [2]
Page 6 arrives and in the frame.

Page fault: 12
ratio: 40.00%
```

缺页错误数为 12，命中率为 40.00%

FIFO 置换算法的流程图如下：



FIFO 算法创建一个队列来管理所有的内存页面，置换的是队列的首个页面。当需要调入页面到内存时，就把它加到队列的尾部。

2) LRU 置换算法

```
reference:
3 6 7 5 3 5 6 2 9 1 2 7 0 9 3 6 0 6 2 6

LRU:

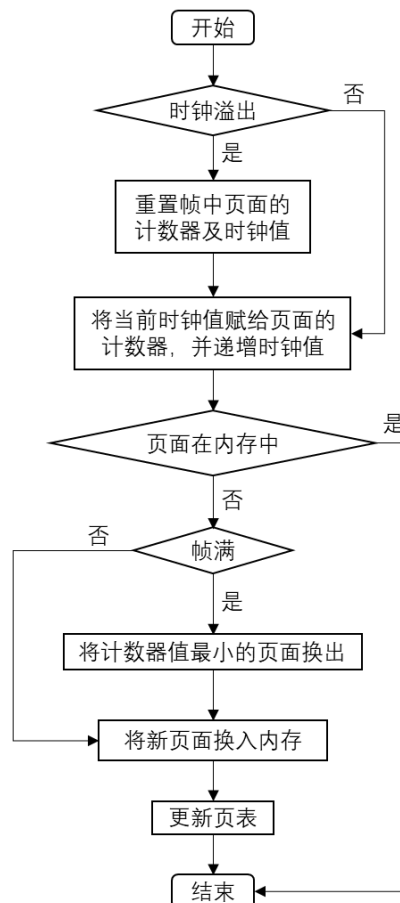
Page 3 arrives but not in the frame.
frame: [3]
Page 6 arrives but not in the frame.
frame: [3] [6]
Page 7 arrives but not in the frame.
frame: [3] [6] [7]
Page 5 arrives but not in the frame.
frame: [3] [6] [7] [5]
Page 3 arrives and in the frame.
Page 5 arrives and in the frame.
Page 6 arrives and in the frame.
Page 2 arrives but not in the frame.
frame: [3] [6] [2] [5]
Page 9 arrives but not in the frame.
frame: [9] [6] [2] [5]
Page 1 arrives but not in the frame.

frame: [9] [6] [2] [1]
Page 2 arrives and in the frame.
Page 7 arrives but not in the frame.
frame: [9] [7] [2] [1]
Page 0 arrives but not in the frame.
frame: [0] [7] [2] [1]
Page 9 arrives but not in the frame.
frame: [0] [7] [2] [9]
Page 3 arrives but not in the frame.
frame: [0] [7] [3] [9]
Page 6 arrives but not in the frame.
frame: [0] [6] [3] [9]
Page 0 arrives and in the frame.
Page 6 arrives and in the frame.
Page 2 arrives but not in the frame.
frame: [0] [6] [3] [2]
Page 6 arrives and in the frame.

Page fault: 13
ratio: 35.00%
```

缺页错误数为 13，命中率为 35.00%

LRU 置换算法（计数器实现）的流程图如下：



LRU 置换将每个页面与它的上次使用的时间关联起来。当需要置换页面时，LRU 选择最长时间没有使用的页面。LRU 算法的一个重要问题是如何确定由上次使用时间定义的帧的顺序，有两个实现是可行的。第一种方案是堆栈方法，即：每当页面被引用时，它就从堆栈中移除并放在顶部。第二种方案是计数器方法，即：为每个页表条目关联一个使用时间域，并为 CPU 添加一个逻辑时钟或计数器，每次内存引用都会递增时钟，并将时钟寄存器的内容复制到对应页表条目的使用时间域。

需要注意的是，当选择计数器实现时，可能会存在时钟溢出的问题。这里的解决方案是，当时钟达到最大值时，将帧中的页面按照各自的计数器大小升序排序，并依次重置计数器为 0、1、2、3……同时，将时钟值重置为帧中的页面个数。

3) Belady 异常

```
reference:
1 2 3 4 1 2 5 1 2 3 4 5
```

FIFO(Belady):

```
Page 1 arrives but not in the frame.
frame: [1]
Page 2 arrives but not in the frame.
frame: [1] [2]
Page 3 arrives but not in the frame.
frame: [1] [2] [3]
Page 4 arrives but not in the frame.
frame: [2] [3] [4]
Page 1 arrives but not in the frame.
frame: [3] [4] [1]
Page 2 arrives but not in the frame.
frame: [4] [1] [2]
Page 5 arrives but not in the frame.
frame: [1] [2] [5]
Page 1 arrives and in the frame.
Page 2 arrives and in the frame.
Page 3 arrives but not in the frame.
frame: [2] [5] [3]
Page 4 arrives but not in the frame.
frame: [5] [3] [4]
Page 5 arrives and in the frame.
```

```
Page fault: 9
ratio: 25.00%
```

```
reference:
1 2 3 4 1 2 5 1 2 3 4 5
```

FIFO(Belady):

```
Page 1 arrives but not in the frame.
frame: [1]
Page 2 arrives but not in the frame.
frame: [1] [2]
Page 3 arrives but not in the frame.
frame: [1] [2] [3]
Page 4 arrives but not in the frame.
frame: [1] [2] [3] [4]
Page 1 arrives and in the frame.
Page 2 arrives and in the frame.
Page 5 arrives but not in the frame.
frame: [2] [3] [4] [5]
Page 1 arrives but not in the frame.
frame: [3] [4] [5] [1]
Page 2 arrives but not in the frame.
frame: [4] [5] [1] [2]
Page 3 arrives but not in the frame.
frame: [5] [1] [2] [3]
Page 4 arrives but not in the frame.
frame: [1] [2] [3] [4]
Page 5 arrives but not in the frame.
frame: [2] [3] [4] [5]
```

```
Page fault: 10
ratio: 16.67%
```

所谓 Belady 异常指的是，对于有些页面置换算法（如 FIFO），随着分配帧数量的增加，缺页错误率可能会增加。

3 帧时，缺页错误数为 9；而 4 帧时，缺页错误数为 10。

3 文件系统

3.1 实验目的

- 1) 通过一个简单文件系统的设计，加深理解文件系统的内部实现原理；
- 2) 了解 EXT2 文件系统的结构；
- 3) 掌握编写复杂代码的分层设计方法。

3.2 实验内容

模拟 EXT2 文件系统原理设计实现一个类 EXT2 文件系统，实现 EXT2 文件系统的功能子集，并用现有操作系统上的文件来代替硬盘进行硬件模拟。具体实现的功能包括：登录 login、修改密码 password、列出当前目录下的内容 ls、创建目录 mkdir、删除目录 rmdir、进入目录 cd、创建文件 create、删除文件 delete、打开文件 open、关闭文件 close、读文件 read、写文件 write、修改文件访问权限 chmod、显示磁盘使用情况 check、格式化磁盘 format、退出 quit。

3.3 实验思想

为了降低文件系统设计的复杂性，可以利用分层思想来组织代码，将文件系统的实现分为 I/O 操作层、文件系统底层、命令层、用户接口层，由下层来为上层提供服务，从而大大简化文件系统设计的难度，并使得代码简单易读。

3.4 实验步骤

(一) I/O 操作层

I/O 操作层的主要任务是完成对磁盘的读写操作。为了完全模拟硬盘读写方式，在类 EXT2 文件系统的实现中，一次只存取 1 个块，即 512 字节。即使只有 1 个字节的修改，也通过读写一个数据块来实现。

通过定义函数 `disk_IO`，传入文件指针偏移量、缓冲区地址、读写控制标识参数，实现在磁盘指定位置读写指定的单个数据块。该函数实际上是对标准 C 库函数 `fseek`、`fread` 及 `fwrite` 的封装。

此外还存在一个问题，缓冲区中的数据是以 `unsigned char` 类型存储的，如何将其转换成指定的数据结构？以 64 字节的 `inode` 为例，通过 `((inode *) Buffer)[i]`，将缓冲区地址强制类型转换为指向 `inode` 的指针，此时就可以将缓冲区 `Buffer` 视为 `inode` 数组，从而对指定位置 `i` 的索引结点进行操作。

(二) 文件系统底层

文件系统底层的主要任务是对单个数据块进行处理，如：在位图中查找空闲位置、输出数据块的内容、在单个数据块中查找文件等。下面简要介绍部分实现细节：

1) 对位图的操作

将位图中的每个字节与掩码 (`0b10000000`) 执行按位与操作，若结果非零则表示掩码中 1 所在位对应的位图位置已分配，此时将掩码右移一位，继续上述操作，直至找到位图中第一个 0 (表示空闲) 的位置。当分配空闲块 (或 `inode`) 或释放已分配的块 (或 `inode`) 时，需要将位图中对应的位执行取反操作。这与寻找位图中的空闲位置类似，只是要先确定待修改的字节位置，并将掩码右移一定的位数，然后执行按位异或操作。

2) 数据块 (或 `inode`) 的分配与释放

分配时要先找到位图中第一个 0 (表示空闲) 的位置，然后将该位取反；释放时则是直接将位图中需要释放的位取反。注意：这两种操作都需更新组描述符。

3) 查看文件是否打开

遍历打开文件表，检查是否存在与该文件相同的 `inode` 号，若存在则表示该文件已打开。

4) 对文件的访问

无论是在当前目录下搜索文件，还是命令层中的读写文件操作等，都需要对文件的所有数据块进行访问。为了最大限度的减少代码量，并实现代码复用，定义函数 `access_file`，该函数有两个参数，第一个是待操作的文件 `inode`，第二个是函数指针 (定义了对单个数据块执行的操作，参数为待操作的数据块号，返回值非零表示继续对文件的访问)。函数 `access_file` 的主体部分是根据文件的数据块个数，通过直接索引、一级间接索引和二级间接索引，将对应的数据块号传递给函数指针。

5) 目录项的创建与删除

由于目录项是变长数据结构，在多次的创建与删除操作后必然有外部碎片的产生。为了

尽可能地利用存储空间，在创建新目录项时，需要先利用已有目录项的 name_len 计算出其实际大小，并与其 rec_len 进行比较，若多余的空间能容纳新建目录项，则修改原有目录项的 rec_len，将其定位到新建目录项的开始位置。在删除目录项时，只需增加前一个目录项的 rec_len 的值，以“跳过”当前这个删除的目录项。

6) 为文件新增数据块

需要对位图进行操作，寻找空闲数据块，并根据文件当前的数据块个数，确定新增数据块应该通过直接索引还是间接索引访问。

(三) 命令层

命令层的主要任务是实现文件系统所支持的指令，如：login、password、ls、mkdir、rmdir、cd 等。为了实现这些命令，本层将使用文件系统底层所提供的服务。下面简要介绍部分实现思路：

1) login

将用户输入的密码与存储在组描述符里的密码进行比较，若匹配则进入文件系统，否则要求用户重新输入密码。

2) password

用户首先输入旧密码，若正确则可输入新密码并确认，否则修改密码失败。

3) ls

遍历当前目录的所有数据块，从目录项中获得文件名及相应的索引结点号，读取 inode 并输出文件的相关信息。

4) mkdir

检查在当前目录下是否存在同名目录，若存在则新建目录失败，否则在当前目录下新建相应的目录项，并为新建的目录分配 inode 与数据块，自动写入当前目录“.”及上一级目录“..”两个目录项。同时，新建目录时还需更新当前目录的最后修改时间。

5) rmdir

检查字符串参数是否为“.”和“..”，若是则删除失败。否则检查在当前目录下是否存在与字符串相匹配的目录，若存在则继续查看该目录是否为空，若为空则删除当前目录下的目录项，并释放目录 inode 及原先分配的数据块。

6) cd

匹配字符串参数，若为“..”则回到上级目录，若为“.”则仍留在当前目录，若为空格或“~”则返回根目录，否则会查找是否存在与字符串匹配的目录，若存在则进入该目录。此外，cd

指令执行成功后需要更改当前索引结点及当前路径名，并修改 cd 指令执行前所在目录的最后访问时间。

7) create

检查在当前目录下是否存在同名文件，若存在则新建失败，否则在当前目录下新建目录项。注意，此时并不为文件分配数据块，只分配了 inode。

8) delete

检查文件是否存在，若存在则继续查看文件的打开状态。若文件已打开，则提示用户先关闭文件再进行删除操作；否则删除当前目录下的目录项，并释放文件 inode 及原先分配的数据块。

9) open

检查文件是否存在，若存在则继续查看文件的打开状态。若文件尚未打开则检查访问权限及可打开文件数是否已达到最大，若符合条件则更新文件打开表。

10) close

检查文件是否存在，若存在则继续查看文件的打开状态。若文件已打开，则将文件打开表中的相应索引结点清零。同时，在关闭文件时还需更新文件的最后访问时间。

11) read

检查文件是否存在，若存在则读取文件 inode，并检查是否有读权限及文件是否已经打开，若符合条件则输出文件所有数据块的内容。

12) write

检查文件是否存在，若存在则读取文件 inode，并检查是否有写权限及文件是否已经打开，若符合条件则定位到文件末尾，并写入用户输入的字符，直至用户输入 SIGQUIT (ctrl+\) 信号结束。在写入文件时，若文件当前的数据块已用完，则在文件最大长度之内为其分配新的数据块。同时，在写文件完成后还需更新文件的最后修改时间。

13) chmod

检查文件是否存在，若存在则根据用户输入的权限码修改文件访问权限。

14) check

读取组描述符，输出磁盘的相关信息及使用情况，包括：卷名、数据块大小、空闲数据块个数、空闲索引结点个数、文件系统当前的目录总数。

15) format

将磁盘的 4611 个块清空，重新写入组描述符，并重新创建根目录。

(四) 用户接口层

用户接口层的主要任务是接收及识别用户命令、词法分析、提取命令及参数、组织调用命令层所对应的命令实现相应功能。本层实际上是一个基于命令层基础上的 shell。

该层的实现思路是：在一个恒为 true 的 while 循环中读取用户输入的字符串，通过 strcmp 函数比较该字符串的前缀是否与文件系统的某个指令相匹配，若匹配则将字符串的后续部分作为参数传递给命令层的相应函数，否则将该字符串视为无效指令。上述操作不断执行，直至用户输入 quit 指令执行 break 退出循环，从而退出文件系统。

3.5 实验运行初值及运行结果分析

1) login

```
guo@guo-virtual-machine:~/exp3$ ./ext2
Password: 666666
*****
Welcome to EXT2 file system!
*****
[root]#
```

图 3-1 login 指令

登录文件系统，进入根目录，初始密码为：666666。注意，'#'左边的方括号内的字符串表示当前所在目录。

2) password

```
[root]# password
Old password: 666666
New password(no more than 9): 123
Confirm password: 123
The password is changed.
[root]#
```

图 3-2 password 指令

使用 password 指令更改密码。

3) mkdir 与 ls

```
[root]# mkdir test
[root]# ls
name      type    mode    size(Byte)  creat time          access time         modify time
.         <DIR>   r_w_    28          Sun Dec  4 09:31:14 2022  Sun Dec  4 09:31:14 2022  Sun Dec  4 09:31:14 2022
..        <DIR>   r_w_    28          Sun Dec  4 09:31:14 2022  Sun Dec  4 09:31:14 2022  Sun Dec  4 09:31:14 2022
test      <DIR>   r_w_    17          Sun Dec  4 09:33:43 2022  Sun Dec  4 09:33:43 2022  Sun Dec  4 09:33:43 2022
[root]#
```

图 3-3 mkdir 与 ls 指令

使用 `mkdir` 指令创建目录 `test`，并使用 `ls` 指令列出当前目录（根目录 `root`）下的文件信息，包括文件名字、类型、访问权限、大小、创建时间、最后访问时间以及最后修改时间。注意到，每个目录文件下都有“.”与“..”两个条目，分别代表当前目录与上一级目录。因此，新创建的目录 `test` 的初始大小即为这两个条目所占空间 17 字节。同时，目录的默认访问权限均为可读可写。

4) `cd`

```
[root]# cd test
[root/test]# ls
name      type    mode    size(Byte)  creat time      access time      modify time
.          <DIR>   r_w_    27           Sun Dec  4 09:33:43 2022  Sun Dec  4 09:36:24 2022  Sun Dec  4 09:33:43 2022
..         <DIR>   r_w_    28           Sun Dec  4 09:31:14 2022  Sun Dec  4 09:33:47 2022  Sun Dec  4 09:31:14 2022
stu        <DIR>   r_w_    17           Sun Dec  4 09:36:28 2022  Sun Dec  4 09:36:28 2022  Sun Dec  4 09:36:28 2022
[root/test]# cd stu
[root/test/stu]# ls
name      type    mode    size(Byte)  creat time      access time      modify time
.          <DIR>   r_w_    17           Sun Dec  4 09:36:28 2022  Sun Dec  4 09:36:28 2022  Sun Dec  4 09:36:28 2022
..         <DIR>   r_w_    27           Sun Dec  4 09:33:43 2022  Sun Dec  4 09:36:44 2022  Sun Dec  4 09:33:43 2022
[root/test/stu]# cd ..
[root/test]# cd
[root]#
```

图 3-4 `cd` 指令

使用 `cd` 指令进入目录，`cd` 后面加目录名进入对应目录，加“..”回到上级目录，加“.”仍留在当前目录，加空格或“~”返回根目录。当目录更改后，“#”号左侧的当前所在目录也会相应更改。

5) `rmdir`

```
[root]# rmdir test
Cannot delete non empty directory!
[root]# cd test
[root/test]# ls
name      type    mode    size(Byte)  creat time      access time      modify time
.          <DIR>   r_w_    27           Sun Dec  4 09:33:43 2022  Sun Dec  4 09:39:59 2022  Sun Dec  4 09:33:43 2022
..         <DIR>   r_w_    28           Sun Dec  4 09:31:14 2022  Sun Dec  4 09:33:47 2022  Sun Dec  4 09:31:14 2022
stu        <DIR>   r_w_    17           Sun Dec  4 09:40:06 2022  Sun Dec  4 09:40:06 2022  Sun Dec  4 09:40:06 2022
[root/test]# rmdir stu
[root/test]# ls
name      type    mode    size(Byte)  creat time      access time      modify time
.          <DIR>   r_w_    17           Sun Dec  4 09:33:43 2022  Sun Dec  4 09:40:21 2022  Sun Dec  4 09:33:43 2022
..         <DIR>   r_w_    28           Sun Dec  4 09:31:14 2022  Sun Dec  4 09:33:47 2022  Sun Dec  4 09:31:14 2022
[root/test]#
```

图 3-5 `rmdir` 指令

使用 `rmdir` 指令删除目录，注意：这里的实现只能删除空目录。因此，在删除非空目录 `test` 时会提示错误信息“Cannot delete non empty directory!”，而空目录 `stu` 则能够正常删除。虽然目录 `stu` 的大小为 17 字节，但它实际存储的是 `stu` 本身以及其上一级目录，因此 `stu` 仍然是一个空目录。

6) create

```
[root]# ls
name      type    mode    size(Byte)  creat time          access time          modify time
.          <DIR>   r_w_    28           Sun Dec  4 09:31:14 2022 Sun Dec  4 09:33:47 2022 Sun Dec  4 09:31:14 2022
..         <DIR>   r_w_    28           Sun Dec  4 09:31:14 2022 Sun Dec  4 09:33:47 2022 Sun Dec  4 09:31:14 2022
test      <DIR>   r_w_    17           Sun Dec  4 09:33:43 2022 Sun Dec  4 09:40:26 2022 Sun Dec  4 09:33:43 2022
[root]# create file
[root]# ls
name      type    mode    size(Byte)  creat time          access time          modify time
.          <DIR>   r_w_    39           Sun Dec  4 09:31:14 2022 Sun Dec  4 09:42:32 2022 Sun Dec  4 09:31:14 2022
..         <DIR>   r_w_    39           Sun Dec  4 09:31:14 2022 Sun Dec  4 09:42:32 2022 Sun Dec  4 09:31:14 2022
test      <DIR>   r_w_    17           Sun Dec  4 09:33:43 2022 Sun Dec  4 09:40:26 2022 Sun Dec  4 09:33:43 2022
file      <FILE>  r_w_x    0            Sun Dec  4 09:42:40 2022 Sun Dec  4 09:42:40 2022 Sun Dec  4 09:42:40 2022
[root]#
```

图 3-6 create 指令

使用 create 指令创建新文件 file。需要注意的是，在模拟的文件系统中，凡是扩展名为.exe, .bin, .com 及不带扩展名的，都被加上 x（可执行）标识。此外，与新建目录不同的是，在新建文件时并不为文件分配数据块，因此文件 file 的大小为 0 字节。只有当对文件进行写操作时，才会真正为文件分配数据块。

7) open 与 write

```
[root]# write file
The file does not open.
[root]# open file
[root]# write file
hello World!!!
GuoSJ
^
[root]# ls
name      type    mode    size(Byte)  creat time          access time          modify time
.          <DIR>   r_w_    39           Sun Dec  4 09:31:14 2022 Sun Dec  4 09:42:43 2022 Sun Dec  4 09:31:14 2022
..         <DIR>   r_w_    39           Sun Dec  4 09:31:14 2022 Sun Dec  4 09:42:43 2022 Sun Dec  4 09:31:14 2022
test      <DIR>   r_w_    17           Sun Dec  4 09:33:43 2022 Sun Dec  4 09:40:26 2022 Sun Dec  4 09:33:43 2022
file      <FILE>  r_w_x    20           Sun Dec  4 09:42:40 2022 Sun Dec  4 09:45:10 2022 Sun Dec  4 09:45:10 2022
[root]#
```

图 3-7 open 与 write 指令

只有在使用 open 指令打开文件后，才能对文件进行写操作，同时会在打开文件表（最多打开 16 个文件）中记录其 inode 号。使用 write 指令写入文件时，输入 SIGQUIT 信号（ctrl+\）作为写操作结束标识。

8) read

```
[root]# read file
hello World!!!
GuoSJ
[root]#
```

图 3-8 read 指令

使用 read 指令读取文件 file，输出结果与写入该文件的字符串相符。

9) close 与 delete

```
[root]# delete file
The file is in use! Please close it first.
[root]# close file
[root]# delete file
[root]# ls
name      type    mode    size(Byte)  creat time          access time          modify time
.          <DIR>   r_w_    28          Sun Dec  4 09:31:14 2022  Sun Dec  4 09:45:18 2022  Sun Dec  4 09:31:14 2022
..         <DIR>   r_w_    28          Sun Dec  4 09:31:14 2022  Sun Dec  4 09:45:18 2022  Sun Dec  4 09:31:14 2022
test      <DIR>   r_w_    17          Sun Dec  4 09:33:43 2022  Sun Dec  4 09:40:26 2022  Sun Dec  4 09:33:43 2022
[root]#
```

图 3-9 close 与 delete 指令

使用 delete 指令删除文件，注意：删除前应先使用 close 指令关闭文件。同时，关闭操作还会将打开文件表中原来的 inode 号置为 0（表示 NULL）。在使用 delete 指令删除文件 file 后，当前目录下已经没有了对应的条目，说明删除成功。

10) chmod

```
[root]# create hhh.txt
[root]# ls
name      type    mode    size(Byte)  creat time          access time          modify time
.          <DIR>   r_w_    42          Sun Dec  4 09:31:14 2022  Sun Dec  4 09:48:16 2022  Sun Dec  4 09:31:14 2022
..         <DIR>   r_w_    42          Sun Dec  4 09:31:14 2022  Sun Dec  4 09:48:16 2022  Sun Dec  4 09:31:14 2022
test      <DIR>   r_w_    17          Sun Dec  4 09:33:43 2022  Sun Dec  4 09:40:26 2022  Sun Dec  4 09:33:43 2022
hhh.txt   <FILE>  r_w_    0           Sun Dec  4 09:49:57 2022  Sun Dec  4 09:49:57 2022  Sun Dec  4 09:49:57 2022
[root]# chmod hhh.txt
modification: 4
[root]# ls
name      type    mode    size(Byte)  creat time          access time          modify time
.          <DIR>   r_w_    42          Sun Dec  4 09:31:14 2022  Sun Dec  4 09:50:00 2022  Sun Dec  4 09:31:14 2022
..         <DIR>   r_w_    42          Sun Dec  4 09:31:14 2022  Sun Dec  4 09:50:00 2022  Sun Dec  4 09:31:14 2022
test      <DIR>   r_w_    17          Sun Dec  4 09:33:43 2022  Sun Dec  4 09:40:26 2022  Sun Dec  4 09:33:43 2022
hhh.txt   <FILE>  r_      0           Sun Dec  4 09:49:57 2022  Sun Dec  4 09:50:46 2022  Sun Dec  4 09:50:46 2022
[root]# open hhh.txt
[root]# write hhh.txt
You do not have permission to write this file.
[root]#
```

图 3-10 chmod 指令

使用 chmod 指令修改文件访问权限。用文件类型码的最低三位分别表示 r(读)、w(写)、x(执行)，置 1 表示允许对应的操作。将文件 hhh.txt 的访问权限修改为 4（即二进制的 100）后，该文件的访问权限变为只读，无法再调用 write 指令对该文件进行写操作。

11) check

```
[root]# check
Volume Name: EXT2FS
Block Size: 512Bytes
Free Block: 4094
Free Inode: 4093
Directories: 2
[root]#
```

图 3-11 check 指令

使用 check 指令查看磁盘使用情况，包括：卷名、数据块大小、空闲数据块个数、空闲索引结点个数、文件系统当前的目录总数。

12) format

```
[root]# ls
name      type    mode    size(Byte)  creat time          access time          modify time
.          <DIR>   r_w_    42           Sun Dec  4 09:31:14 2022  Sun Dec  4 09:50:48 2022  Sun Dec  4 09:31:14 2022
..         <DIR>   r_w_    42           Sun Dec  4 09:31:14 2022  Sun Dec  4 09:50:48 2022  Sun Dec  4 09:31:14 2022
test      <DIR>   r_w_    17           Sun Dec  4 09:33:43 2022  Sun Dec  4 09:40:26 2022  Sun Dec  4 09:33:43 2022
hhh.txt   <FILE>  r_      0            Sun Dec  4 09:49:57 2022  Sun Dec  4 09:50:57 2022  Sun Dec  4 09:50:46 2022

[root]# format
Format succeeded!
Volume Name: EXT2FS
Block Size: 512Bytes
Free Block: 4095
Free Inode: 4095
Directories: 1
[root]# ls
name      type    mode    size(Byte)  creat time          access time          modify time
.          <DIR>   r_w_    17           Sun Dec  4 09:54:35 2022  Sun Dec  4 09:54:35 2022  Sun Dec  4 09:54:35 2022
..         <DIR>   r_w_    17           Sun Dec  4 09:54:35 2022  Sun Dec  4 09:54:35 2022  Sun Dec  4 09:54:35 2022

[root]#
```

图 3-12 format 指令

使用 format 指令格式化磁盘。该操作将磁盘的位图与数据块清空，重新写入组描述符，并重新创建根目录。

13) quit

```
[root]# quit
guo@guo-virtual-machine:~/exp3$
```

图 3-13 quit 指令

使用 quit 指令退出文件系统。

3.6 实验总结

3.6.1 实验中的问题与解决过程

1) 如何对文件（虚拟磁盘）进行读写操作；

使用 fopen、fclose 打开与关闭文件，使用 fseek 定位文件指针的位置，使用 fread、fwrite 对文件进行读写。需要注意的是，给函数 fopen 传入参数“rb+”时对应的文件必须存在，否则返回空指针；而传入参数“wb+”时若文件不存在则会新建一个文件，若文件存在则会清空文件。

2) 如何处理变长的目录项；

存取：将数据块读取到字节数组中后，将指向字节数组的指针强制类型转换为指向 dir_entry，此时就可以对目录项进行操作；写入操作同理。此外，为了防止写入时溢出，将

数据块的缓冲区定义为 1024B，但实际存储的有效大小为前 512B。

定位：rec_len 记录了目录项的大小，将它与当前目录的起始地址相加，就得到了下一个有效目录的起始地址。要删除当前这个目录，只需要将 inode 置为 0，并增加前一个目录项的 rec_len 的值，以“跳过”当前这个删除的目录。

同时，发现目录项结构体 dir_entry 的大小应为 261B，但使用 sizeof 运算符得出的实际大小却是 262B，猜测是因为字节对齐。

参考资料：<https://blog.csdn.net/qwertyupoiuytr/article/details/70471623>

3) 如何判断写操作的结束：

最开始的想法是使用 EOF（Windows 下是 ctrl+z，Linux 下是 ctrl+d）作为写操作的结束标识。此方法在 Windows 下运行良好，但在 Linux 下输入 EOF 后会导致后续的所有输入函数失效，程序异常终止。

在网上查找资料的过程中，找到了一个类似的提问，但并未给出解决方案。详见：

<https://stackoverflow.com/questions/57648464/>

最终的解决方案：使用软中断，自定义 SIGQUIT (ctrl+\) 信号处理函数，用以退出写操作循环。

4) 如何 debug。

在对文件（虚拟磁盘）进行读写时，可能会把数据写入到错误的块中，或是从错误的块中读取了数据，此时单凭检查程序难以发现问题所在。可以借助于软件 Hex Editor Neo 以 16 进制模式查看文件，观察存储在文件（虚拟磁盘）中的二进制串，方便定位错误。

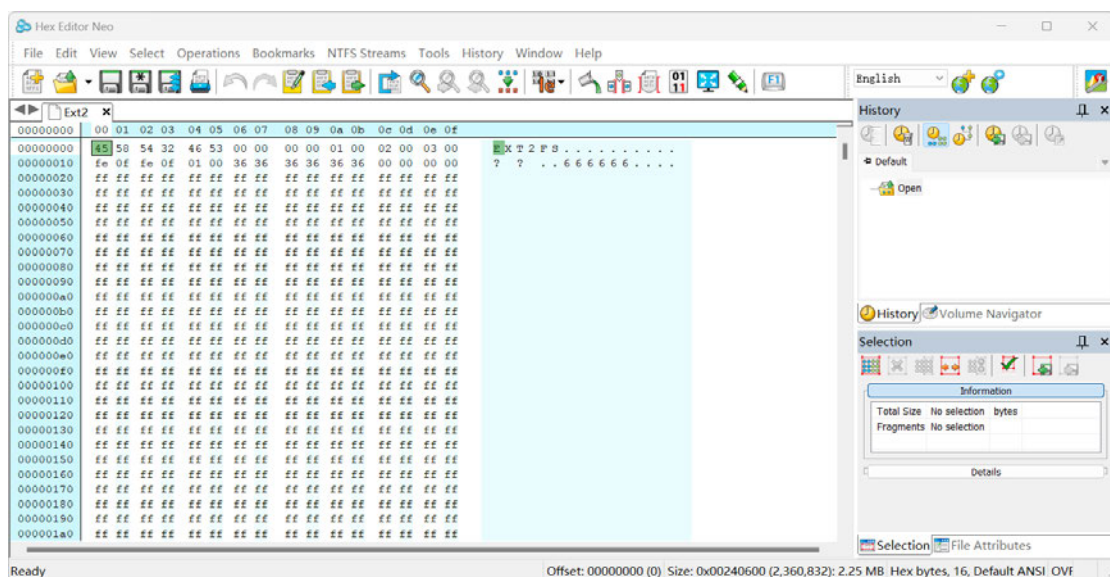


图 3-14 利用 Hex Editor Neo 进行 Debug

3.6.2 实验收获

通过本次实验，我了解了 EXT2 文件系统的数据结构与实现原理，并实际动手设计了一个类 EXT2 文件系统，实现了一些文件系统的基本功能，同时还掌握了编写复杂代码的分层设计方法。总之，本次实验让我受益匪浅。

3.6.3 意见与建议

建议老师增加集中答疑的时间，回答同学们在实验过程中遇到的问题。

3.7 附件

3.7.1 附件 1 程序

//类EXT2文件系统：
//逻辑块与物理块大小均为512B，只有一个用户，只定义一个组，省略超级块

```
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <signal.h>

#define VOLUME_NAME "EXT2FS"           //卷名
#define EXT2_NAME_LEN 255             //文件名最大长度
#define FOPEN_TABLE_MAX 16            //文件打开表最大大小
#define BLOCK_SIZE 512                 //数据块大小
#define DATA_BLOCK_COUNT 4096         //数据块个数
#define INODE_COUNT 4096               //inode个数
#define DISK_SIZE 4611                 //磁盘块个数
#define READ_DISK 1                    //读磁盘
#define WRITE_DISK 0                   //写磁盘
#define GDT_START 0                    //组描述符起始偏移
#define BLOCK_BITMAP_START 512         //数据块位图起始偏移
#define INODE_BITMAP_START 1024        //inode位图起始偏移
#define INODE_TABLE_START 1536         //inode表起始偏移
#define DATA_BLOCK_START (1536+512*512) //数据块起始偏移

//文件类型
enum {
    FT_UNKNOWN,           //未知
    FT_REG_FILE,          //普通文件
    FT_DIR,               //目录
    FT_CHRDEV,            //字符设备
    FT_BLKDEV,            //块设备
    FT_FIFO,              //管道
    FT_SOCKET,            //套接字
    FT_SYMLINK,           //符号指针
};
```

```

//组描述符(32B)
typedef struct group_desc {
    char bg_volume_name[10];           //卷名
    unsigned short bg_block_bitmap;    //保存数据块位图的块号
    unsigned short bg_inode_bitmap;    //保存inode位图的块号
    unsigned short bg_inode_table;     //inode表的起始块号
    unsigned short bg_free_blocks_count; //本组空闲块个数
    unsigned short bg_free_inodes_count; //本组空闲inode个数
    unsigned short bg_used_dirs_count;  //本组目录个数
    char bg_password[10];              //登录密码
} group_desc;

//索引结点(64B)
typedef struct inode {
    unsigned short i_mode;              //文件类型及访问权限
    unsigned short i_blocks;            //文件的数据块个数
    unsigned int i_size;                //文件或目录的大小(字节)
    unsigned int i_atime;               //访问时间
    unsigned int i_ctime;               //创建时间
    unsigned int i_mtime;               //修改时间
    unsigned int i_dtime;               //删除时间
    unsigned short i_block[8];          //指向数据块的指针
    char i_pad[24];                    //填充(0xff)
} inode;

//目录项(变长7-261B)
typedef struct dir_entry {
    unsigned short inode;               //索引结点号
    unsigned short rec_len;              //目录项长度
    unsigned char name_len;              //文件名长度
    unsigned char file_type;             //文件类型
    char name[EXT2_NAME_LEN];           //文件名
} dir_entry;

//读写缓冲区
group_desc gdt;                        //组描述符缓冲区
inode inode_buf;                       //索引结点缓冲区
dir_entry dir_buf;                     //目录项缓冲区
unsigned char block_bitmap[BLOCK_SIZE]; //块位图缓冲区
unsigned char inode_bitmap[BLOCK_SIZE]; //inode位图缓冲区
unsigned char gdt_Buffer[BLOCK_SIZE];   //组描述符读写缓冲区
unsigned char inode_Buffer[BLOCK_SIZE]; //索引结点数据块缓冲区
unsigned char Buffer[BLOCK_SIZE * 2];   //数据块缓冲区

/* Buffer的后半部分并不存储实际数据, 只是用来防止写入数据时溢出 */

FILE *fp = NULL;                       //虚拟磁盘
char current_path[256];                 //当前路径(字符串)
unsigned short current_dir;              //当前目录(索引结点)
unsigned short fopen_table[FOPEN_TABLE_MAX]; //文件打开表
char search_file_name[EXT2_NAME_LEN];    //待查找的文件名

```



```
/******IO操作*****/

//磁盘I/O (以块为单位)
void disk_IO(long offset, void *buf, int rw_flag) {
    fseek(fp, offset, SEEK_SET);
    if (rw_flag) {
        fread(buf, BLOCK_SIZE, 1, fp);
    } else {
        fwrite(buf, BLOCK_SIZE, 1, fp);
        fflush(fp);
    }
}

//载入组描述符
void load_group_desc(void) {
    disk_IO(GDT_START, gdt_Buffer, READ_DISK);
    gdt = ((group_desc *)gdt_Buffer)[0];
}

//更新组描述符
void update_group_desc(void) {
    memset(gdt_Buffer, 0xff, sizeof(gdt_Buffer));
    ((group_desc *)gdt_Buffer)[0] = gdt;
    disk_IO(GDT_START, gdt_Buffer, WRITE_DISK);
}

/* inode从1开始计数, 0表示NULL; 数据块从0开始计数 */

//载入第k个inode
void load_inode_entry(unsigned short k) {
    --k;
    unsigned short i = k / 8, j = k % 8;
    disk_IO(INODE_TABLE_START + i * BLOCK_SIZE, inode_Buffer, READ_DISK);
    inode_buf = ((inode *)inode_Buffer)[j];
}

//更新第k个inode
void update_inode_entry(unsigned short k) {
    --k;
    unsigned short i = k / 8, j = k % 8;
    disk_IO(INODE_TABLE_START + i * BLOCK_SIZE, inode_Buffer, READ_DISK);
    ((inode *)inode_Buffer)[j] = inode_buf;
    disk_IO(INODE_TABLE_START + i * BLOCK_SIZE, inode_Buffer, WRITE_DISK);
}

//载入第i个数据块
void load_block_entry(unsigned short i) {
    disk_IO(DATA_BLOCK_START + i * BLOCK_SIZE, Buffer, READ_DISK);
}

//更新第i个数据块
void update_block_entry(unsigned short i) {
    disk_IO(DATA_BLOCK_START + i * BLOCK_SIZE, Buffer, WRITE_DISK);
}
```

```
//载入数据块位图
void load_block_bitmap(void) {
    disk_IO(BLOCK_BITMAP_START, block_bitmap, READ_DISK);
}

//更新数据块位图
void update_block_bitmap(void) {
    disk_IO(BLOCK_BITMAP_START, block_bitmap, WRITE_DISK);
}

//载入inode位图
void load_inode_bitmap(void) {
    disk_IO(INODE_BITMAP_START, inode_bitmap, READ_DISK);
}

//更新inode位图
void update_inode_bitmap(void) {
    disk_IO(INODE_BITMAP_START, inode_bitmap, WRITE_DISK);
}

/*****底层*****/

//返回位图中的第一个0的位置
int bitmap_find_0(unsigned char bitmap[]) {
    for (int i = 0; i < BLOCK_SIZE; ++i) {
        unsigned char mask = 0b10000000;
        if (bitmap[i] == 0b11111111)
            continue;
        for (int j = 0; j < 8; ++j) {
            if (!(bitmap[i] & mask))
                return i * 8 + j;
            mask >>= 1;
        }
    }
    return -1;
}

//将位图中的第k位取反
void bitmap_neg_k(unsigned char bitmap[], int k) {
    int i = k / 8, j = k % 8;
    unsigned char mask = 0b10000000;
    for (int t = 0; t < j; ++t)
        mask >>= 1;
    bitmap[i] ^= mask;
}

//初始化inode
void initialize_inode(void) {
    inode_buf.i_mode = FT_UNKNOWN;
    inode_buf.i_mode <<= 8;
    //默认访问权限: 可读可写不可执行
    ((unsigned char *)(&(inode_buf.i_mode)))[1] = 0b00000110;
    inode_buf.i_blocks = 0;
    inode_buf.i_size = 0;
    inode_buf.i_atime = time(NULL);
}
```

```
inode_buf.i_ctime = time(NULL);
inode_buf.i_mtime = time(NULL);
inode_buf.i_dtime = 0;
memset(inode_buf.i_block, 0, sizeof(inode_buf.i_block));
memset(inode_buf.i_pad, 0xff, sizeof(inode_buf.i_pad));
}

//分配inode
unsigned short new_inode(void) {
    load_group_desc();
    if (gdt.bg_free_inodes_count) {
        //更新inode位图
        load_inode_bitmap();
        unsigned short i = bitmap_find_0(inode_bitmap);
        bitmap_neg_k(inode_bitmap, i);
        update_inode_bitmap();
        //更新inode表
        initialize_inode();
        update_inode_entry(i + 1);
        //更新组描述符
        gdt.bg_free_inodes_count--;
        update_group_desc();
        return i + 1;
    } else {
        printf("There is no inode to be allocated!\n");
        return 0;
    }
}

//释放inode
void free_inode(unsigned short i) {
    //更新inode位图
    load_inode_bitmap();
    bitmap_neg_k(inode_bitmap, i - 1);
    update_inode_bitmap();
    //更新组描述符
    load_group_desc();
    gdt.bg_free_inodes_count++;
    update_group_desc();
}

//分配数据块
unsigned short new_block(void) {
    load_group_desc();
    if (gdt.bg_free_blocks_count) {
        //更新数据块位图
        load_block_bitmap();
        unsigned short i = bitmap_find_0(block_bitmap);
        bitmap_neg_k(block_bitmap, i);
        update_block_bitmap();
        //更新组描述符
        gdt.bg_free_blocks_count--;
        update_group_desc();
        return i;
    } else {
        printf("There is no block to be allocated!\n");
    }
}
```

```
        return 0;
    }
}

//释放数据块
void free_block(unsigned short i) {
    //更新数据块位图
    load_block_bitmap();
    bitmap_neg_k(block_bitmap, i);
    update_block_bitmap();
    //更新组描述符
    load_group_desc();
    gdt.bg_free_blocks_count++;
    update_group_desc();
}

//创建目录项
void new_dir_entry(char file_name[], unsigned char file_type) {
    memset(&dir_buf, 0, sizeof(dir_entry));
    dir_buf.inode = new_inode();
    dir_buf.name_len = strlen(file_name);
    dir_buf.rec_len = 7 + dir_buf.name_len;
    dir_buf.file_type = file_type;
    strcpy(dir_buf.name, file_name);
    //更新inode文件类型
    load_inode_entry(dir_buf.inode);
    ((unsigned char *)(&(inode_buf.i_mode)))[0] = file_type;
    char *extension = strchr(file_name, '.');
    if (!extension || !strcmp(extension, ".exe") || !strcmp(extension, ".bin")
        || !strcmp(extension, ".com"))
        ((unsigned char *)(&(inode_buf.i_mode)))[1] = 0b000000111;
    update_inode_entry(dir_buf.inode);
    //建立目录时
    if (file_type == FT_DIR) {
        //更新组描述符
        load_group_desc();
        gdt.bg_used_dirs_count++;
        update_group_desc();
        //分配数据块
        unsigned short i = new_block();
        memset(Buffer, 0, sizeof(Buffer));
        dir_entry temp;
        //创建当前目录项
        memset(&temp, 0, sizeof(dir_entry));
        temp.inode = dir_buf.inode;
        temp.rec_len = 8;
        temp.name_len = 1;
        temp.file_type = FT_DIR;
        strcpy(temp.name, ".");
        ((dir_entry *)Buffer)[0] = temp;
        //创建上一级目录项
        memset(&temp, 0, sizeof(dir_entry));
        temp.inode = current_dir;
        temp.rec_len = 9;
        temp.name_len = 2;
        temp.file_type = FT_DIR;
```

```

        strcpy(temp.name, "..");
        ((dir_entry *) (Buffer + 8))[0] = temp;
        //更新数据块
        update_block_entry(i);
        //更新inode
        load_inode_entry(dir_buf.inode);
        inode_buf.i_blocks = 1;
        inode_buf.i_size = 17;
        inode_buf.i_block[0] = i;
        ((unsigned char *) (&(inode_buf.i_mode)))[1] = 0b00000110;
        update_inode_entry(dir_buf.inode);
    }
}

//查看文件是否打开
unsigned short is_open(unsigned short inode_num) {
    for (int i = 0; i < FOPEN_TABLE_MAX; ++i)
        if (fopen_table[i] == inode_num)
            return 1;
    return 0;
}

//访问文件的所有数据块
unsigned short access_file(unsigned short inode_num, unsigned short (*func)(unsigned short)) {
    //func对一个数据块进行操作，当返回值非零时，退出对文件的访问
    load_inode_entry(inode_num);
    unsigned short indirect_1 = 0, indirect_2 = 0;
    for (unsigned short i = 0; i < inode_buf.i_blocks; /**/) {
        if (i < 6) {
            //直接索引
            load_block_entry(inode_buf.i_block[i]);
            unsigned short ret = func(inode_buf.i_block[i]);
            update_block_entry(inode_buf.i_block[i]);
            ++i;
            if (ret) return ret;
        } else if (i == 6) {
            //一级间接索引
            load_block_entry(inode_buf.i_block[i]);
            unsigned short j;
            if (indirect_1 < BLOCK_SIZE / sizeof(unsigned short))
                j = ((unsigned short *) Buffer)[indirect_1];
            if (j == 0 || indirect_1 == BLOCK_SIZE / sizeof(unsigned short)) {
                ++i;
                indirect_1 = 0;
                continue;
            }
            ++indirect_1;
            load_block_entry(j);
            unsigned short ret = func(j);
            update_block_entry(j);
            if (ret) return ret;
        } else {
            //二级间接索引
            load_block_entry(inode_buf.i_block[i]);
            unsigned short j, k;

```

```

        if (indirect_1 < BLOCK_SIZE / sizeof(unsigned short))
            j = ((unsigned short *)Buffer)[indirect_1];
        if (j == 0 || indirect_1 == BLOCK_SIZE / sizeof(unsigned short))
            break;
        load_block_entry(j);
        if (indirect_2 < BLOCK_SIZE / sizeof(unsigned short))
            k = ((unsigned short *)Buffer)[indirect_2];
        if (k == 0)
            break;
        if (indirect_2 == BLOCK_SIZE / sizeof(unsigned short)) {
            ++indirect_1;
            indirect_2 = 0;
            continue;
        }
        ++indirect_2;
        load_block_entry(k);
        unsigned short ret = func(k);
        update_block_entry(k);
        if (ret) return ret;
    }
}
return 0;
}

//在单个数据块中查找空闲位置并写入目录项
unsigned short search_free_dir_in_block(unsigned short block) {
    unsigned short current_pos = 0;
    memset(Buffer + BLOCK_SIZE, 0, BLOCK_SIZE);
    dir_entry temp = ((dir_entry *)Buffer)[0];
    do {
        unsigned short k = temp.rec_len - temp.name_len - 7;
        if (k >= dir_buf.rec_len) {
            temp.rec_len = temp.name_len + 7;
            ((dir_entry *) (Buffer + current_pos))[0].rec_len = temp.rec_len;
            dir_buf.rec_len = k;
            ((dir_entry *) (Buffer + current_pos + temp.rec_len))[0].inode =
            dir_buf.inode;
            ((dir_entry *) (Buffer + current_pos + temp.rec_len))[0].rec_len =
            dir_buf.rec_len;
            ((dir_entry *) (Buffer + current_pos + temp.rec_len))[0].name_len =
            dir_buf.name_len;
            ((dir_entry *) (Buffer + current_pos + temp.rec_len))[0].file_type =
            dir_buf.file_type;
            strcpy(((dir_entry *) (Buffer + current_pos + temp.rec_len))[0].name,
            dir_buf.name);
            return 1;
        }
        current_pos += temp.rec_len;
        temp = ((dir_entry *) (Buffer + current_pos))[0];
    } while (temp.inode);
    if (BLOCK_SIZE - current_pos > dir_buf.rec_len) {
        ((dir_entry *) (Buffer + current_pos))[0] = dir_buf;
        return 1;
    }
    return 0;
}

```

//在单个数据块中查找文件

```
unsigned short search_in_block(unsigned short block) {
    unsigned short current_pos = 0;
    //确保存在 dir_buf.inode == 0
    memset(Buffer + BLOCK_SIZE, 0, BLOCK_SIZE);
    dir_buf = ((dir_entry *)Buffer)[0];
    do {
        if (!strcmp(dir_buf.name, search_file_name))
            return dir_buf.inode;
        current_pos += dir_buf.rec_len;
        dir_buf = ((dir_entry *) (Buffer + current_pos))[0];
    } while (dir_buf.inode);
    return 0;
}
```

//在当前目录下查找文件

```
unsigned short search_file(char name[]) {
    strcpy(search_file_name, name);
    return access_file(current_dir, search_in_block);
}
```

//在单个数据块中删除目录项

```
unsigned short delete_in_block(unsigned short block) {
    unsigned short current_pos = 0, pre_pos = 0;
    //确保存在 dir_buf.inode == 0
    memset(Buffer + BLOCK_SIZE, 0, BLOCK_SIZE);
    dir_buf = ((dir_entry *)Buffer)[0];
    do {
        if (!strcmp(dir_buf.name, search_file_name)) {
            ((dir_entry *) (Buffer + pre_pos))[0].rec_len += dir_buf.rec_len;
            return dir_buf.inode;
        }
        pre_pos = current_pos;
        current_pos += dir_buf.rec_len;
        dir_buf = ((dir_entry *) (Buffer + current_pos))[0];
    } while (dir_buf.inode);
    return 0;
}
```

//释放文件数据块

```
unsigned short free_file_block(unsigned short block) {
    free_block(block);
    return 0;
}
```

//输出文件单个数据块的内容

```
unsigned short print_file(unsigned short block) {
    for (unsigned short i = 0; i < BLOCK_SIZE; ++i) {
        if (Buffer[i] == 0)
            return 1;
        putchar(Buffer[i]);
    }
    return 0;
}
```

//输出目录单个数据块的内容

```

unsigned short print_dir(unsigned short block) {
    inode inode_temp = inode_buf;
    unsigned short current_pos = 0;
    memset(Buffer + BLOCK_SIZE, 0, BLOCK_SIZE);
    dir_buf = ((dir_entry *)Buffer)[0];
    do {
        load_inode_entry(dir_buf.inode);
        printf("%-10s", dir_buf.name);
        switch (dir_buf.file_type) {
            case FT_DIR:
                printf("<DIR> ");
                break;
            case FT_REG_FILE:
                printf("<FILE> ");
                break;
            default:
                printf("unknown ");
        }
        switch (((unsigned char *)&(inode_buf.i_mode))[1]) {
            case 2:
                printf("__w__ ");
                break;
            case 4:
                printf("r____ ");
                break;
            case 6:
                printf("r_w__ ");
                break;
            case 7:
                printf("r_w_x ");
                break;
            default:
                printf("error ");
        }
        printf("%-12hu", inode_buf.i_size);
        time_t temp;
        char time_str[26];
        temp = inode_buf.i_ctime;
        strcpy(time_str, ctime(&temp));
        time_str[24] = '\0';
        printf("%s ", time_str);
        temp = inode_buf.i_atime;
        strcpy(time_str, ctime(&temp));
        time_str[24] = '\0';
        printf("%s ", time_str);
        temp = inode_buf.i_mtime;
        strcpy(time_str, ctime(&temp));
        time_str[24] = '\0';
        printf("%s \n", time_str);
        current_pos += dir_buf.rec_len;
        dir_buf = ((dir_entry *) (Buffer + current_pos))[0];
    } while (dir_buf.inode);
    inode_buf = inode_temp;
    return 0;
}

```


//为文件新增数据块

```

unsigned short add_file_block(unsigned short inode_num) {
    unsigned short i = 0;
    load_inode_entry(inode_num);
    if (inode_buf.i_blocks < 6) {
        i = new_block();
        inode_buf.i_blocks++;
        inode_buf.i_block[inode_buf.i_blocks - 1] = i;
    } else if (inode_buf.i_blocks == 6) {
        i = new_block();
        inode_buf.i_blocks++;
        inode_buf.i_block[inode_buf.i_blocks - 1] = i;
        i = new_block();
        memset(Buffer, 0, sizeof(Buffer));
        ((unsigned short *)Buffer)[0] = i;
        update_block_entry(inode_buf.i_block[inode_buf.i_blocks - 1]);
    } else if (inode_buf.i_blocks == 7) {
        load_block_entry(inode_buf.i_block[6]);
        unsigned short j;
        for (j = 0; j < 256 && ((unsigned short *)Buffer)[j]; ++j);
        if (j == 256) {
            i = new_block();
            inode_buf.i_blocks++;
            inode_buf.i_block[inode_buf.i_blocks - 1] = i;
            i = new_block();
            memset(Buffer, 0, sizeof(Buffer));
            ((unsigned short *)Buffer)[0] = i;
            update_block_entry(inode_buf.i_block[inode_buf.i_blocks - 1]);
            unsigned short k = new_block();
            memset(Buffer, 0, sizeof(Buffer));
            ((unsigned short *)Buffer)[0] = k;
            update_block_entry(i);
            i = k;
        } else {
            i = new_block();
            ((unsigned short *)Buffer)[j] = i;
        }
        update_block_entry(inode_buf.i_block[6]);
    } else {
        load_block_entry(inode_buf.i_block[7]);
        unsigned short j, k;
        for (j = 0; j < 256 && ((unsigned short *)Buffer)[j]; ++j);
        --j;
        load_block_entry(j);
        for (k = 0; k < 256 && ((unsigned short *)Buffer)[k]; ++k);
        if (k < 256) {
            i = new_block();
            ((unsigned short *)Buffer)[k] = i;
            update_block_entry(j);
        } else {
            if (j < 255) {
                load_block_entry(inode_buf.i_block[7]);
                ++j;
                unsigned short temp = new_block();
                ((unsigned short *)Buffer)[j] = temp;
            }
        }
    }
}

```

```

        update_block_entry(inode_buf.i_block[7]);
        memset(Buffer, 0, sizeof(Buffer));
        i = new_block();
        ((unsigned short *)Buffer)[0] = i;
        update_block_entry(temp);
    } else
        printf("The file has reached the maximum capacity!\n");
    }
}
update_inode_entry(inode_num);
return i;
}

```

//gets_s()函数是C11的编译器扩展项，用于替换不安全的gets()
 //在Linux的gcc中并不支持该函数
 //可使用fgets()来实现指定个数字符的读取，但也会把行尾的换行符读入
 //因此需要用空字符来替换换行符

```

char *gets_s(char *buffer, int num) {
    if (fgets(buffer, num, stdin) != 0) {
        size_t len = strlen(buffer);
        if (len > 0 && buffer[len - 1] == '\n')
            buffer[len - 1] = '\0';
        return buffer;
    }
    return NULL;
}

```

/******初始化文件系统******/

//初始化内存数据

```

void initialize_memory(void) {
    //清空文件打开表
    memset(fopen_table, 0, sizeof(fopen_table));
    //初始化组描述符
    strcpy(gdt.bg_volume_name, VOLUME_NAME);
    strcpy(gdt.bg_password, "666666");
    gdt.bg_block_bitmap = 1;
    gdt.bg_inode_bitmap = 2;
    gdt.bg_inode_table = 3;
    gdt.bg_free_blocks_count = DATA_BLOCK_COUNT;
    gdt.bg_free_inodes_count = INODE_COUNT;
    gdt.bg_used_dirs_count = 0;
    //初始化当前路径
    current_dir = 1;
    strcpy(current_path, "root");
}

```

//初始化磁盘

```

void initialize_disk(void) {
    //清空磁盘
    memset(Buffer, 0, sizeof(Buffer));
    for (int i = 0; i < DISK_SIZE; ++i)
        disk_IO(i * BLOCK_SIZE, Buffer, WRITE_DISK);
    //将组描述符写入磁盘
    update_group_desc();
}

```

```
//创建根目录
new_dir_entry("root", FT_DIR);
}

/*****命令层*****/

//登录
unsigned short login(char password[]) {
    load_group_desc();
    return !(strcmp(gdt.bg_password, password));
}

//修改密码
void change_password(void) {
    printf("Old password: ");
    char password[10];
    gets_s(password, 9);
    if (login(password)) {
        printf("New password(no more than 9): ");
        gets_s(password, 9);
        char confirm[10];
        printf("Confirm password: ");
        gets_s(confirm, 9);
        if (!strcmp(password, confirm)) {
            load_group_desc();
            strcpy(gdt.bg_password, password);
            update_group_desc();
            printf("The password is changed.\n");
        } else
            printf("Please try again.\n");
    } else
        printf("Password error!\n");
}

//列出当前目录
void dir(void) {
    printf("name      ");
    printf("type      ");
    printf("mode      ");
    printf("size(Byte)  ");
    printf("creat time      ");
    printf("access time      ");
    printf("modify time      \n");
    access_file(current_dir, print_dir);
    load_inode_entry(current_dir);
    inode_buf.i_atime = time(NULL);
    update_inode_entry(current_dir);
}

//建立目录
void mkdir(char name[]) {
    unsigned short i = search_file(name);
    if (i) {
        printf("A directory with the same name exists.\n");
    } else {
```

```

    new_dir_entry(name, FT_DIR);
    //在当前目录下添加新建目录项
    if (!access_file(current_dir, search_free_dir_in_block)) {
        //数据块不够时
        unsigned short j = add_file_block(i);
        memset(Buffer, 0, sizeof(Buffer));
        ((dir_entry *)Buffer)[0] = dir_buf;
        update_block_entry(j);
    }
    load_inode_entry(current_dir);
    inode_buf.i_size += (7 + strlen(name));
    update_inode_entry(current_dir);
}

//删除空目录
void rmdir(char name[]) {
    if ((!strcmp(name, ".") || (!strcmp(name, "..")))) {
        printf("Wrong command!\n");
        return;
    }
    unsigned short i = search_file(name);
    if (i) {
        load_inode_entry(i);
        if (((unsigned char *)&(inode_buf.i_mode))[0] != FT_DIR) {
            printf("Wrong command!\n");
            return;
        }
        if (inode_buf.i_size != 17) {
            printf("Cannot delete non empty directory!\n");
            return;
        }
        //删除当前目录下的目录项
        access_file(current_dir, delete_in_block);
        //释放inode及数据块
        access_file(i, free_file_block);
        load_inode_entry(i);
        inode_buf.i_dtime = time(NULL);
        update_inode_entry(i);
        free_inode(i);
        //更新当前目录大小
        load_inode_entry(current_dir);
        inode_buf.i_size -= (7 + strlen(name));
        update_inode_entry(current_dir);
        //更新组描述符
        load_group_desc();
        gdt.bg_used_dirs_count--;
        update_group_desc();
    } else {
        printf("The directory does not exist.\n");
    }
}

//建立文件
void create(char name[]) {
    unsigned short i = search_file(name);

```

```
if (i) {
    printf("A file with the same name exists.\n");
} else {
    new_dir_entry(name, FT_REG_FILE);
    //在当前目录下添加新建目录项
    if (!access_file(current_dir, search_free_dir_in_block)) {
        //数据块不够时
        unsigned short j = add_file_block(i);
        memset(Buffer, 0, sizeof(Buffer));
        ((dir_entry *)Buffer)[0] = dir_buf;
        update_block_entry(j);
    }
    load_inode_entry(current_dir);
    inode_buf.i_size += (7 + strlen(name));
    update_inode_entry(current_dir);
}
}

//删除文件
void delete (char name[]) {
    unsigned short i = search_file(name);
    if (i) {
        load_inode_entry(i);
        if (((unsigned char *)&(inode_buf.i_mode))[0] != FT_REG_FILE) {
            printf("Wrong command!\n");
            return;
        }
        if (is_open(i))
            printf("The file is in use! Please close it first.\n");
        else {
            //删除当前目录下的目录项
            access_file(current_dir, delete_in_block);
            //释放inode及数据块
            access_file(i, free_file_block);
            load_inode_entry(i);
            inode_buf.i_dtime = time(NULL);
            update_inode_entry(i);
            free_inode(i);
            //更新当前目录大小
            load_inode_entry(current_dir);
            inode_buf.i_size -= (7 + strlen(name));
            update_inode_entry(current_dir);
        }
    } else {
        printf("The file does not exist.\n");
    }
}

//切换单级目录
void cd(char path[]) {
    if (!strcmp(path, "") || !strcmp(path, "~")) {
        current_dir = 1;
        strcpy(current_path, "root");
    } else if (!strcmp(path, ".")) {
        ; //do nothing
    } else if (!strcmp(path, "..")) {

```

```

        load_inode_entry(current_dir);
        load_block_entry(inode_buf.i_block[0]);
        current_dir = ((dir_entry *) (Buffer + 8))[0].inode;
        for (int k = strlen(current_path); k >= 0; --k) {
            if (current_path[k] == '/') {
                current_path[k] = '\0';
                break;
            }
        }
    } else {
        unsigned short i = search_file(path);
        if (i) {
            load_inode_entry(i);
            if (((unsigned char *) (&(inode_buf.i_mode)))[0] != FT_DIR) {
                printf("No such directory exists!\n");
                return;
            }
            current_dir = i;
            strcat(current_path, "/");
            strcat(current_path, path);
        } else {
            printf("No such directory exists!\n");
        }
    }
}

//更改文件保护码
void attrib(char name[], unsigned char change) {
    unsigned short i = search_file(name);
    if (i == 0)
        printf("The file does not exist.\n");
    else {
        if (change == 2 || change == 4 || change == 6 || change == 7) {
            load_inode_entry(i);
            (((unsigned char *) (&(inode_buf.i_mode)))[1] = change;
            inode_buf.i_atime = time(NULL);
            inode_buf.i_mtime = time(NULL);
            update_inode_entry(i);
        } else
            printf("Wrong modification!\n");
    }
}

//打开文件
void open(char name[]) {
    unsigned short inode_num = search_file(name);
    if (inode_num == 0)
        printf("The file does not exist.\n");
    else {
        if (is_open(inode_num))
            printf("The file has opened.\n");
        else {
            load_inode_entry(inode_num);
            unsigned char permission = (((unsigned char *) (&(inode_buf.i_mode)))[1];
            if (permission == 4 || permission == 6 || permission == 7) {
                inode_buf.i_atime = time(NULL);
            }
        }
    }
}

```

```
        update_inode_entry(inode_num);
        for (unsigned short i = 0; i < FOPEN_TABLE_MAX; ++i) {
            if (fopen_table[i] == 0) {
                fopen_table[i] = inode_num;
                return;
            }
        }
        printf("The number of files opened has reached the maximum.\n");
    } else
        printf("You do not have permission to open this file.\n");
    }
}
```

//关闭文件

```
void close(char name[]) {
    unsigned short inode_num = search_file(name);
    if (inode_num == 0)
        printf("The file does not exist.\n");
    else {
        if (is_open(inode_num)) {
            load_inode_entry(inode_num);
            inode_buf.i_atime = time(NULL);
            update_inode_entry(inode_num);
            for (unsigned short i = 0; i < FOPEN_TABLE_MAX; ++i) {
                if (fopen_table[i] == inode_num) {
                    fopen_table[i] = 0;
                    return;
                }
            }
        } else
            printf("The file does not open.\n");
    }
}
```

//读文件

```
void read(char name[]) {
    unsigned short i = search_file(name);
    if (i == 0)
        printf("The file does not exist.\n");
    else {
        load_inode_entry(i);
        unsigned char permission = ((unsigned char *)&(inode_buf.i_mode))[1];
        if (permission == 4 || permission == 6 || permission == 7) {
            if (is_open(i)) {
                access_file(i, print_file);
                inode_buf.i_atime = time(NULL);
                update_inode_entry(i);
            } else
                printf("The file does not open.\n");
        } else
            printf("You do not have permission to read this file.\n");
    }
}
```

```
//写文件（以附加方式）
int flag = 1;    //用于退出写操作
void stopWrite() {
    flag = 0;
}
void write(char name[]) {
    unsigned short i = search_file(name);
    if (i == 0)
        printf("The file does not exist.\n");
    else {
        load_inode_entry(i);
        unsigned char permission = ((unsigned char *)(&(inode_buf.i_mode)))[1];
        if (permission == 2 || permission == 6 || permission == 7) {
            if (is_open(i)) {
                //定位文件末尾
                unsigned short pos = 0, j = 0;
                if (inode_buf.i_blocks == 0) {
                    j = new_block();
                    inode_buf.i_blocks = 1;
                    inode_buf.i_block[0] = j;
                    update_inode_entry(i);
                    load_block_entry(j);
                    pos = 0;
                } else if (inode_buf.i_blocks <= 6) {
                    j = inode_buf.i_block[inode_buf.i_blocks - 1];
                    load_block_entry(j);
                    pos = inode_buf.i_size % BLOCK_SIZE;
                    if (pos == 0) {
                        j = add_file_block(i);
                        load_block_entry(j);
                    }
                } else if (inode_buf.i_blocks == 7) {
                    load_block_entry(inode_buf.i_block[6]);
                    pos = inode_buf.i_size / BLOCK_SIZE - 6;
                    j = ((unsigned short *)Buffer)[pos];
                    load_block_entry(j);
                    pos = inode_buf.i_size % BLOCK_SIZE;
                    if (pos == 0) {
                        j = add_file_block(i);
                        load_block_entry(j);
                    }
                } else {
                    load_block_entry(inode_buf.i_block[7]);
                    pos = (inode_buf.i_size / BLOCK_SIZE - 6 - 256) / 256;
                    load_block_entry(((unsigned short *)Buffer)[pos]);
                    pos = (inode_buf.i_size / BLOCK_SIZE - 6 - 256) % 256;
                    j = ((unsigned short *)Buffer)[pos];
                    load_block_entry(j);
                    pos = inode_buf.i_size % BLOCK_SIZE;
                    if (pos == 0) {
                        j = add_file_block(i);
                        load_block_entry(j);
                    }
                }
            }
            //写入数据块
            unsigned short new_size = 0;
```



```

char ch = getchar();
//在Windows中可使用EOF (ctrl+z) 来作为文件输入流的结束
//但在Linux中使用EOF (ctrl+d) 时, 则会出现意想不到的错误
//在Linux中输入EOF会导致后续的所有输入函数失效
//因此, 在Linux下可通过软中断来实现写操作的结束
//通过定义SIGQUIT (ctrl+\) 信号的处理函数来退出写操作循环
signal(SIGQUIT, stopWrite);
while (flag/* ch != EOF */) {
    Buffer[pos] = ch;
    pos++;
    new_size++;
    ch = getchar();
    if (pos == BLOCK_SIZE) {
        update_block_entry(j);
        j = add_file_block(i);
        load_block_entry(j);
        pos = 0;
    }
    flag = 1;
    Buffer[pos] = '\0';
    update_block_entry(j);
    inode_buf.i_size += new_size - 1;
    inode_buf.i_atime = time(NULL);
    inode_buf.i_mtime = time(NULL);
    update_inode_entry(i);
} else
    printf("The file does not open.\n");
} else
    printf("You do not have permission to write this file.\n");
}
}

//显示磁盘信息
void check_disk(void) {
    load_group_desc();
    printf("Volume Name: %s\n", gdt.bg_volume_name);
    printf("Block Size: %dBytes\n", BLOCK_SIZE);
    printf("Free Block: %u\n", gdt.bg_free_blocks_count);
    printf("Free Inode: %u\n", gdt.bg_free_inodes_count);
    printf("Directories: %u\n", gdt.bg_used_dirs_count);
}

//格式化
void format(void) {
    initialize_memory();
    initialize_disk();
    printf("Format succeeded!\n");
    check_disk();
}

/*****用户接口层*****/

//用户接口
void shell(void) {

```

```
char cmd[256] = "";
while (1) {
    printf("[%s]# ", current_path);
    gets_s(cmd, 256);
    if (!strcmp(cmd, "format")) {
        format();
    } else if (!strcmp(cmd, "check")) {
        check_disk();
    } else if (!strcmp(cmd, "password")) {
        change_password();
    } else if (!strcmp(cmd, "ls")) {
        dir();
    } else if (!strncmp(cmd, "mkdir ", 6)) {
        mkdir(cmd + 6);
    } else if (!strncmp(cmd, "rmdir ", 6)) {
        rmdir(cmd + 6);
    } else if (!strncmp(cmd, "create ", 7)) {
        create(cmd + 7);
    } else if (!strncmp(cmd, "delete ", 7)) {
        delete (cmd + 7);
    } else if (!strncmp(cmd, "cd ", 3)) {
        cd(cmd + 3);
    } else if (!strncmp(cmd, "chmod ", 6)) {
        printf("modification: ");
        unsigned char change;
        scanf("%hhu", &change);
        getchar();
        attrib(cmd + 6, change);
    } else if (!strncmp(cmd, "open ", 5)) {
        open(cmd + 5);
    } else if (!strncmp(cmd, "close ", 6)) {
        close(cmd + 6);
    } else if (!strncmp(cmd, "read ", 5)) {
        read(cmd + 5);
    } else if (!strncmp(cmd, "write ", 6)) {
        write(cmd + 6);
    } else if (!strcmp(cmd, "quit")) {
        break;
    } else {
        printf("Wrong command!\n");
    }
}

int main(void) {
    fp = fopen("./Ext2", "rb+");
    initialize_memory();
    if (fp == NULL) {
        fp = fopen("./Ext2", "wb+");
        initialize_disk();
    }
    printf("Password: ");
    char password[10];
    gets_s(password, 9);
    while (!login(password)) {
        printf("Error!\n");
    }
}
```

```

        printf("Password: ");
        gets_s(password, 9);
    }
    printf("*****\n");
    printf("    Welcome to EXT2 file system!\n");
    printf("*****\n");
    shell();
    fclose(fp);

    return 0;
}

```

3.7.2 附件 2 Readme

一、程序运行结果

1) login

```

guo@guo-virtual-machine:~/exp3$ ./ext2
Password: 666666
*****
    Welcome to EXT2 file system!
*****
[root]#

```

登录文件系统，进入根目录，初始密码为：666666。注意，'#'左边的方括号内的字符串表示当前所在目录。

2) password

```

[root]# password
Old password: 666666
New password(no more than 9): 123
Confirm password: 123
The password is changed.
[root]#

```

使用 password 指令更改密码。

3) mkdir 与 ls

```

[root]# mkdir test
[root]# ls

```

name	type	mode	size(Byte)	creat time	access time	modify time
.	<DIR>	r_w__	28	Sun Dec 4 09:31:14 2022	Sun Dec 4 09:31:14 2022	Sun Dec 4 09:31:14 2022
..	<DIR>	r_w__	28	Sun Dec 4 09:31:14 2022	Sun Dec 4 09:31:14 2022	Sun Dec 4 09:31:14 2022
test	<DIR>	r_w__	17	Sun Dec 4 09:33:43 2022	Sun Dec 4 09:33:43 2022	Sun Dec 4 09:33:43 2022

```

[root]#

```

使用 mkdir 指令创建目录 test，并使用 ls 指令列出当前目录（根目录 root）下的文件信息，包括文件名字、类型、访问权限、大小、创建时间、最后访问时间以及最后修改时间。注意到，每个目录文件下都有“.”与“..”两个条目，分别代表当前目录与上一级目录。因此，新创建的目录 test 的初始大小即为这两个条目所占空间 17 字节。同时，目录的默认访问权限均为可读可写。

4) cd

```
[root]# cd test
[root/test]# ls
name      type    mode    size(Byte)  creat time          access time          modify time
.          <DIR>   r_w_    27           Sun Dec  4 09:33:43 2022  Sun Dec  4 09:36:24 2022  Sun Dec  4 09:33:43 2022
..         <DIR>   r_w_    28           Sun Dec  4 09:31:14 2022  Sun Dec  4 09:33:47 2022  Sun Dec  4 09:31:14 2022
stu        <DIR>   r_w_    17           Sun Dec  4 09:36:28 2022  Sun Dec  4 09:36:28 2022  Sun Dec  4 09:36:28 2022
[root/test]# cd stu
[root/test/stu]# ls
name      type    mode    size(Byte)  creat time          access time          modify time
.          <DIR>   r_w_    17           Sun Dec  4 09:36:28 2022  Sun Dec  4 09:36:28 2022  Sun Dec  4 09:36:28 2022
..         <DIR>   r_w_    27           Sun Dec  4 09:33:43 2022  Sun Dec  4 09:36:44 2022  Sun Dec  4 09:33:43 2022
[root/test/stu]# cd ..
[root/test]# cd
[root]#
```

使用 cd 指令进入目录，cd 后面加目录名进入对应目录，加“..”回到上级目录，加“.”仍留在当前目录，加空格或“~”返回根目录。当目录更改后，‘#’号左侧的当前所在目录也会相应更改。

5) rmdir

```
[root]# rmdir test
Cannot delete non empty directory!
[root]# cd test
[root/test]# ls
name      type    mode    size(Byte)  creat time          access time          modify time
.          <DIR>   r_w_    27           Sun Dec  4 09:33:43 2022  Sun Dec  4 09:39:59 2022  Sun Dec  4 09:33:43 2022
..         <DIR>   r_w_    28           Sun Dec  4 09:31:14 2022  Sun Dec  4 09:33:47 2022  Sun Dec  4 09:31:14 2022
stu        <DIR>   r_w_    17           Sun Dec  4 09:40:06 2022  Sun Dec  4 09:40:06 2022  Sun Dec  4 09:40:06 2022
[root/test]# rmdir stu
[root/test]# ls
name      type    mode    size(Byte)  creat time          access time          modify time
.          <DIR>   r_w_    17           Sun Dec  4 09:33:43 2022  Sun Dec  4 09:40:21 2022  Sun Dec  4 09:33:43 2022
..         <DIR>   r_w_    28           Sun Dec  4 09:31:14 2022  Sun Dec  4 09:33:47 2022  Sun Dec  4 09:31:14 2022
[root/test]#
```

使用 rmdir 指令删除目录，注意：这里的实现只能删除空目录。因此，在删除非空目录 test 时会提示错误信息“Cannot delete non empty directory!”，而空目录 stu 则能够正常删除。虽然目录 stu 的大小为 17 字节，但它实际存储的是 stu 本身以及其上一级目录，因此 stu 仍然是一个空目录。

6) create

```
[root]# ls
name      type    mode    size(Byte)  creat time          access time          modify time
.          <DIR>   r_w_    28           Sun Dec  4 09:31:14 2022  Sun Dec  4 09:33:47 2022  Sun Dec  4 09:31:14 2022
..         <DIR>   r_w_    28           Sun Dec  4 09:31:14 2022  Sun Dec  4 09:33:47 2022  Sun Dec  4 09:31:14 2022
test       <DIR>   r_w_    17           Sun Dec  4 09:33:43 2022  Sun Dec  4 09:40:26 2022  Sun Dec  4 09:33:43 2022
[root]# create file
[root]# ls
name      type    mode    size(Byte)  creat time          access time          modify time
.          <DIR>   r_w_    39           Sun Dec  4 09:31:14 2022  Sun Dec  4 09:42:32 2022  Sun Dec  4 09:31:14 2022
..         <DIR>   r_w_    39           Sun Dec  4 09:31:14 2022  Sun Dec  4 09:42:32 2022  Sun Dec  4 09:31:14 2022
test       <DIR>   r_w_    17           Sun Dec  4 09:33:43 2022  Sun Dec  4 09:40:26 2022  Sun Dec  4 09:33:43 2022
file       <FILE>  r_w_x    0           Sun Dec  4 09:42:40 2022  Sun Dec  4 09:42:40 2022  Sun Dec  4 09:42:40 2022
[root]#
```

使用 `create` 指令创建新文件 `file`。需要注意的是，在模拟的文件系统中，凡是扩展名为 `.exe`、`.bin`、`.com` 及不带扩展名的，都被加上 `x`（可执行）标识。此外，与新建目录不同的是，在新建文件时并不为文件分配数据块，因此文件 `file` 的大小为 0 字节。只有当对文件进行写操作时，才会真正为文件分配数据块。

7) `open` 与 `write`

```
[root]# write file
The file does not open.
[root]# open file
[root]# write file
hello World!!!
GuoSJ
^
[root]# ls
```

name	type	mode	size(Byte)	creat time	access time	modify time
.	<DIR>	r_w__	39	Sun Dec 4 09:31:14 2022	Sun Dec 4 09:42:43 2022	Sun Dec 4 09:31:14 2022
..	<DIR>	r_w__	39	Sun Dec 4 09:31:14 2022	Sun Dec 4 09:42:43 2022	Sun Dec 4 09:31:14 2022
test	<DIR>	r_w__	17	Sun Dec 4 09:33:43 2022	Sun Dec 4 09:40:26 2022	Sun Dec 4 09:33:43 2022
file	<FILE>	r_w_x	20	Sun Dec 4 09:42:40 2022	Sun Dec 4 09:45:10 2022	Sun Dec 4 09:45:10 2022

```
[root]#
```

只有在使用 `open` 指令打开文件后，才能对文件进行写操作，同时会在打开文件表（最多打开 16 个文件）中记录其 `inode` 号。使用 `write` 指令写入文件时，输入 `SIGQUIT` 信号（`ctrl+\`）作为写操作结束标识。

8) `read`

```
[root]# read file
hello World!!!
GuoSJ
[root]#
```

使用 `read` 指令读取文件 `file`，输出结果与写入该文件的字符串相符。

9) `close` 与 `delete`

```
[root]# delete file
The file is in use! Please close it first.
[root]# close file
[root]# delete file
[root]# ls
```

name	type	mode	size(Byte)	creat time	access time	modify time
.	<DIR>	r_w__	28	Sun Dec 4 09:31:14 2022	Sun Dec 4 09:45:18 2022	Sun Dec 4 09:31:14 2022
..	<DIR>	r_w__	28	Sun Dec 4 09:31:14 2022	Sun Dec 4 09:45:18 2022	Sun Dec 4 09:31:14 2022
test	<DIR>	r_w__	17	Sun Dec 4 09:33:43 2022	Sun Dec 4 09:40:26 2022	Sun Dec 4 09:33:43 2022

```
[root]#
```

使用 `delete` 指令删除文件，注意：删除前应先使用 `close` 指令关闭文件。同时，关闭操作还会将打开文件表中原来的 `inode` 号置为 0（表示 `NULL`）。在使用 `delete` 指令删除文件 `file` 后，当前目录下已经没有了对应的条目，说明删除成功。

10) `chmod`

```
[root]# create hhh.txt
[root]# ls
name      type    mode    size(Byte)  creat time          access time          modify time
.         <DIR>   r_w__   42          Sun Dec  4 09:31:14 2022  Sun Dec  4 09:48:16 2022  Sun Dec  4 09:31:14 2022
..        <DIR>   r_w__   42          Sun Dec  4 09:31:14 2022  Sun Dec  4 09:48:16 2022  Sun Dec  4 09:31:14 2022
test      <DIR>   r_w__   17          Sun Dec  4 09:33:43 2022  Sun Dec  4 09:40:26 2022  Sun Dec  4 09:33:43 2022
hhh.txt   <FILE>  r_w__   0           Sun Dec  4 09:49:57 2022  Sun Dec  4 09:49:57 2022  Sun Dec  4 09:49:57 2022
[root]# chmod hhh.txt
modification: 4
[root]# ls
name      type    mode    size(Byte)  creat time          access time          modify time
.         <DIR>   r_w__   42          Sun Dec  4 09:31:14 2022  Sun Dec  4 09:50:00 2022  Sun Dec  4 09:31:14 2022
..        <DIR>   r_w__   42          Sun Dec  4 09:31:14 2022  Sun Dec  4 09:50:00 2022  Sun Dec  4 09:31:14 2022
test      <DIR>   r_w__   17          Sun Dec  4 09:33:43 2022  Sun Dec  4 09:40:26 2022  Sun Dec  4 09:33:43 2022
hhh.txt   <FILE>  r_____ 0           Sun Dec  4 09:49:57 2022  Sun Dec  4 09:50:46 2022  Sun Dec  4 09:50:46 2022
[root]# open hhh.txt
[root]# write hhh.txt
You do not have permission to write this file.
[root]#
```

使用 chmod 指令修改文件访问权限。用文件类型码的最低三位分别表示 r(读)、w(写)、x(执行)，置 1 表示允许对应的操作。将文件 hhh.txt 的访问权限修改为 4 (即二进制的 100) 后，该文件的访问权限变为只读，无法再调用 write 指令对该文件进行写操作。

11) check

```
[root]# check
Volume Name: EXT2FS
Block Size: 512Bytes
Free Block: 4094
Free Inode: 4093
Directories: 2
[root]#
```

使用 check 指令查看磁盘使用情况，包括：卷名、数据块大小、空闲数据块个数、空闲索引结点个数、文件系统当前的目录总数。

12) format

```
[root]# ls
name      type    mode    size(Byte)  creat time          access time          modify time
.         <DIR>   r_w__   42          Sun Dec  4 09:31:14 2022  Sun Dec  4 09:50:48 2022  Sun Dec  4 09:31:14 2022
..        <DIR>   r_w__   42          Sun Dec  4 09:31:14 2022  Sun Dec  4 09:50:48 2022  Sun Dec  4 09:31:14 2022
test      <DIR>   r_w__   17          Sun Dec  4 09:33:43 2022  Sun Dec  4 09:40:26 2022  Sun Dec  4 09:33:43 2022
hhh.txt   <FILE>  r_____ 0           Sun Dec  4 09:49:57 2022  Sun Dec  4 09:50:57 2022  Sun Dec  4 09:50:46 2022
[root]# format
Format succeeded!
Volume Name: EXT2FS
Block Size: 512Bytes
Free Block: 4095
Free Inode: 4095
Directories: 1
[root]# ls
name      type    mode    size(Byte)  creat time          access time          modify time
.         <DIR>   r_w__   17          Sun Dec  4 09:54:35 2022  Sun Dec  4 09:54:35 2022  Sun Dec  4 09:54:35 2022
..        <DIR>   r_w__   17          Sun Dec  4 09:54:35 2022  Sun Dec  4 09:54:35 2022  Sun Dec  4 09:54:35 2022
[root]#
```

使用 format 指令格式化磁盘。该操作将磁盘的位图与数据块清空，重新写入组描述符，并重新创建根目录。

13) quit

```
[root]# quit
guo@guo-virtual-machine:~/exp3$
```

使用 quit 指令退出文件系统。

二、遇到的问题

1) 如何对文件（虚拟磁盘）进行读写操作；

使用 fopen、fclose 打开与关闭文件，使用 fseek 定位文件指针的位置，使用 fread、fwrite 对文件进行读写。需要注意的是，给函数 fopen 传入参数“rb+”时对应的文件必须存在，否则返回空指针；而传入参数“wb+”时若文件不存在则会新建一个文件，若文件存在则会清空文件。

2) 如何处理变长的目录项；

存取：将数据块读取到字节数组中后，将指向字节数组的指针强制类型转换为指向 dir_entry，此时就可以对目录项进行操作；写入操作同理。此外，为了防止写入时溢出，将数据块的缓冲区定义为 1024B，但实际存储的有效大小为前 512B。

定位：rec_len 记录了目录项的大小，将它与当前目录的起始地址相加，就得到了下一个有效目录的起始地址。要删除当前这个目录，只需要将 inode 置为 0，并增加前一个目录项的 rec_len 的值，以“跳过”当前这个删除的目录。

同时，发现目录项结构体 dir_entry 的大小应为 261B，但使用 sizeof 运算符得出的实际大小却是 262B，猜测是因为字节对齐。

参考资料：<https://blog.csdn.net/qwertyupoiuytr/article/details/70471623>

3) 如何判断写操作的结束；

最开始的想法是使用 EOF（Windows 下是 ctrl+z，Linux 下是 ctrl+d）作为写操作的结束标识。此方法在 Windows 下运行良好，但在 Linux 下输入 EOF 后会导致后续的所有输入函数失效，程序异常终止。

在网上查找资料的过程中，找到了一个类似的提问，但并未给出解决方案。详见：
<https://stackoverflow.com/questions/57648464/>

最终的解决方案：使用软中断，自定义 SIGQUIT (ctrl+\) 信号处理函数，用以退出写操作循环。

4) 如何 debug。

在对文件（虚拟磁盘）进行读写时，可能会把数据写入到错误的块中，或是从错误的块

中读取了数据，此时单凭检查程序难以发现问题所在。可以借助于软件 Hex Editor Neo 以 16 进制模式查看文件，观察存储在文件（虚拟磁盘）中的二进制串，方便定位错误。

