

实验七：UNIX V6++ 文件系统

1. 实验目的

结合课程所学知识,通过在 UNIX V6++实验环境中编写使用文件管理相关的系统调用或库函数的应用程序,进一步了解 UNIX 文件管理的工作过程。

2. 实验设备及工具

已配置好 UNIX V6++运行和调试环境的 PC 机一台。

3. 预备知识

- (1) 在 UNIX V6++的/lib/file.c 文件中了解 UNIX V6++支持的所有和文件管理有关的库函数。
- (2) 复习利用 fork, wait 和 exit 如何进行多进程编程及父子进程间的同步。
- (3) 熟悉 UNIX 文件系统的内存打开结构和父子进程对文件打开结构的共享。

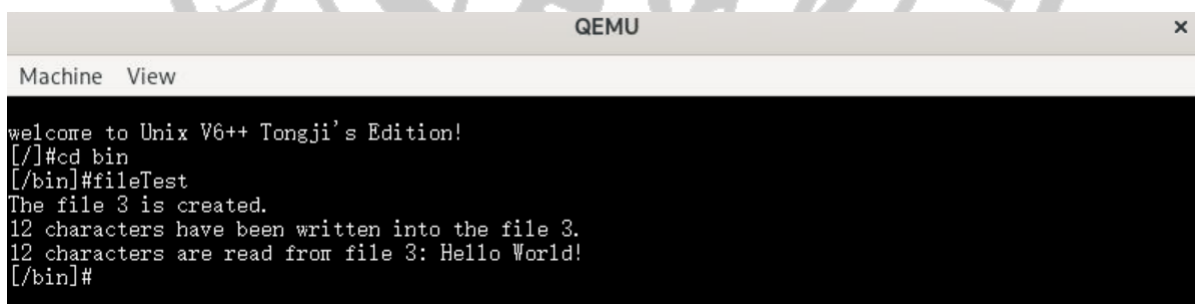
4. 实验内容

4.1. 熟悉 UNIX 文件系统的接口

请读者利用掌握的 UNIX 文件系统接口,在 UNIX V6++中编写可执行程序 fileText,实现以下功能:

- (1) 进程在根目录下创建文件“/Jessy”,创建时设置三类用户对该文件都有读写和可执行的权限;
- (2) 向其中写入字符串“Hello World!”
- (3) 进程将“/Jessy”文件的内容读出,屏幕打印,以判断写入的是否正确。

代码 1 给出了实现上述功能的参考代码。图 1 示出了该参考代码的输出结果。



```
QEMU x
Machine View
welcome to Unix V6++ Tongji's Edition!
[/]#cd bin
[/bin]#fileTest
The file 3 is created.
12 characters have been written into the file 3.
12 characters are read from file 3: Hello World!
[/bin]#
```

图 1: 代码 1 执行结果

根据代码 1 和图 1 的输出,请读者回答以下问题:



```
QEMU x
Machine View
welcome to Unix V6++ Tongji's Edition!
[/]#cd bin
[/bin]#fileTest
The file 3 is created.
12 characters have been written into the file 3.
-1 characters are read from file 3:
[/bin]#
```

图 2: 不重新打开文件直接读的输出结果

```

#include <stdio.h>
#include <sys.h>
#include <file.h>

void main1()
{
    char data1[13]="Hello World!";
    char data2[13];
    int fd = 0;
    int count = 0;

    fd = creat("Jessy",0666);
    if (fd>0)
    {
        printf("The file %d is created.\n",fd);
    }
    else
    {
        printf("The file can not be created.\n");
    }

    count = write(fd, data1, 13);
    if (count == 13)
    {
        printf("%d characters have been written into the file %d.\n", count,fd);
    }
    else
    {
        printf("The file can not be written successfully.\n");
    }
    close(fd);

    fd = open("Jessy",3);           //以可读可写的方式打开文件
    count = read(fd, data2, 12);
    printf("%d characters are read from file %d: %s.", count, fd, data2);
    printf("\n");
    close(fd);
}

```

代码 1

(1) 文件创建成功之后，为什么没有直接完成读写操作，而是写过之后，先关闭，再重新打开？读者可以尝试在代码 1 中将写操作完成之后的关闭文件和打开文件两句代码注释掉，将得到如图 2 所示的错误输出。尝试解释其中的原因，并从 UNIX V6++ 的代码中找到依据。

(2) 在代码 1 中，将两个字符串数组的长度都改为 12，如下所示：

```

char data1[12]="Hello World!";
char data2[12];

```

程序运行将获得如图 4 所示的输出，请解释出现这样的输出的原因。



```

QEMU
Machine View
welcome to Unix V6++ Tongji's Edition!
[/]/#cd bin
[/bin]#fileTest
The file 3 is created.
12 characters have been written into the file 3.
12 characters are read from file 3: Hello World!Hello World!.
[/bin]#

```

图 4：两个字符串数组的长度都改为 12 后的输出

4.2. 父子进程共享文件的读写权限和读写指针

将 fileTest 程序的代码修改成如代码 2 所示。代码主要流程如下：

- (1) 父进程首先创建了“/Jessy”文件，创建时，给三类用户分别设置了读写和可执行的权限；
- (2) 创建成功后，父进程将该文件关闭；
- (3) 父进程以可读可写的权限重新打开该文件，此时，建立了该文件的内存打开结构；
- (4) 成功创建子进程后，父进程睡眠等待子进程结束；
- (5) 子进程上台后，通过共享的文件打开结构，向“/Jessy”中写入“Hello World!”，子进程结束，

唤醒父进程；

- (6) 父进程上台后，从该文件中读出“Hello World!”，并在屏幕打印。

程序运行后，得到和 4.1 节中图 1 相同的输出，如 5 所示。请读者回答以下问题：

- (1) 以文字或绘制的方式说明在代码 2 中，父子进程如果实现对文件的读写权限和读写指针的共享；
- (2) 父进程被唤醒重新上台后，为什么要执行 seek 语句，如果没有这条语句，程序最后的输出是什么样的？为什么？
- (3) 代码 2 中，父子进程执行的 close 操作有何不同？

4.3. 父子进程以不同的读写权限打开文件

在本节实验中，要求读者按以下要求编写程序，并得到和图 5 完全一样的输出：

- (1) 由父进程创建磁盘文件“/Jessy”，创建时为三类用户分别设置可读可写和可执行的权限；
- (2) 父进程创建子进程，并睡眠等待子进程结束；
- (3) 子进程上台后，以可写的方式打开该文件，并向其中写入字符串“Hello World!”，关闭文件，进程终止，并将写入的字符个数以终止码的方式传递给父进程；
- (4) 父进程被唤醒重新上台后，以只读的方式打开该文件，按照子进程终止码的数量，从该文件中读取字符，并在屏幕打印，关闭文件。

程序正确运行后，请读者回答下列问题：

- (1) 以文字或绘制的方式说明在你的代码中，父子进程对 Jessy 文件的共享方式；
- (2) 在这样的共享方式中，父进程被唤醒重新上台后，是否还需要执行 seek 函数，为什么？
- (3) 此处父子进程关闭文件的操作有何不同？

```
#include <stdio.h>
#include <sys.h>
#include <file.h>

void main1()
{
    char data1[13]="Hello World!";
    char data2[13];
    int fd = 0;
    int count = 0;
    int i,j;

    fd = creat("Jessy",0666);        //刚创建好的文件，访问方式是可写
    if (fd>0)
    {
        printf("The file %d is created.\n",fd);
    }
    else
    {
        printf("The file can not be created.\n");
    }
    close(fd);

    fd = open("Jessy",3);            //以可读可写的方式打开文件

    if(fork())
    {
        i=wait(&j);
        seek(fd,0,0);
        count = read(fd, data2, 12);
        printf("%d characters are read from file %d: %s.", count, fd, data2);
        printf("\n");
        close(fd);
    }
    else
    {
        count = write(fd, data1, 13);
        if (count == 13)
        {
            printf("%d characters have been written into the file %d.\n", count,fd);
        }
        else
        {
            printf("The file can not be written successfully.\n");
        }
        close(fd);
        exit(0);
    }
}
```



```
QEMU
Machine View
welcome to Unix V6++ Tongji's Edition!
[/]#cd bin
[/bin]#fileTest
The file 3 is created.
12 characters have been written into the file 3.
12 characters are read from file 3: Hello World!
[/bin]#
```

图 5: 代码 2 的输出结果

5. 实验报告要求

本次实验报告需完成以下内容:

- (1) (1 分) 完成实验 4.1, 并回答其中的 2 个问题。
- (2) (1 分) 完成实验 4.2, 并回答其中的 3 个问题。
- (3) (2 分) 完成实验 4.3, 并回答其中的 2 个问题。

参考答案：

实验 4.1 中的 2 个问题：

(1) 文件创建成功之后，默认以可写的方式打开文件，所以这时候，继续对文件进行写操作是没有问题的，但是进行读操作会因为没有相应的权限而无法进行。如下图所示，可以看到，在打开文件的 `Creat` 函数的代码中，在文件创建成功后，以 `File::FWRITE` 的权限，即写权限，打开文件，等待写入。

```
/*
 * 功能：创建一个新的文件
 * 效果：建立打开文件结构，内存 i 节点开锁、i_count 为正数（应该是 1）
 */
void FileManager::Creat()
{
    Inode* pInode;
    User& u = Kernel::Instance().GetUser();
    unsigned int newACCMODE = u.u_arg[1] & (Inode::IRWXU|Inode::IRWXG|Inode::IRWXO);

    /* 搜索目录的模式为 1，表示创建；若父目录不可写，出错返回 */
    pInode = this->NameI(NextChar, FileManager::CREATE);
    /* 没有找到相应的 Inode，或 NameI 出错 */
    if ( NULL == pInode )
    {
        if(u.u_error)
            return;
        /* 创建 Inode */
        pInode = this->MakNode( newACCMODE & (~Inode::ISVTX) );
        /* 创建失败 */
        if ( NULL == pInode )
        {
            return;
        }

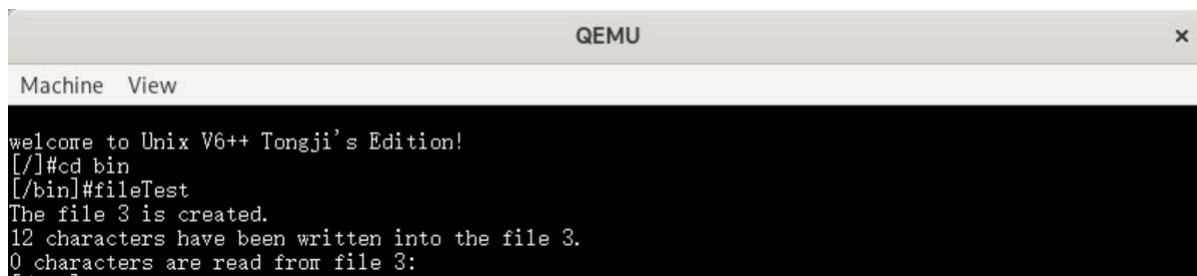
        /*
         * 如果所希望的名字不存在，使用参数 trf = 2 来调用 open1()。
         * 不需要进行权限检查，因为刚刚建立的文件的权限和传入参数 mode
         * 所表示的权限内容是一样的。
         */
        this->Open1(pInode, File::FWRITE, 2);
    }
    else
    {
        /* 如果 NameI() 搜索到已经存在要创建的文件，则清空该文件（用算法 ITrunc()）。UID 没有改变
         * 原来 UNIX 的设计是这样：文件看上去就像新建的文件一样。然而，新文件所有者和许可权方式没变。
         * 也就是说 creat 指定的 RWX 比特无效。
         * 邓蓉认为这是不合理的，应该改变。
         * 现在的实现：creat 指定的 RWX 比特有效 */
        this->Open1(pInode, File::FWRITE, 1);
        pInode->i_mode |= newACCMODE;
    }
}
```

(2) 在代码 1 中, 如果将两个字符串数组的长度都改为 12, 则字符串没有结束符, 而在 main1 函数的栈帧中, data1 紧跟 data2 存储, 因而在最后一个 printf 语句处, 将 data2 和 data1 连续打印出来。

实验 4.2 中的 3 个问题:

(1) 在代码 2 中, 父子进程共享了同一个 File 结构和同一个内存 iNode 节点, 因为具有相同的文件读写权限和文件读写指针。

(2) 因为父子进程共享同一个 File 结构, 因为使用同一个文件读写指针。在子进程完成文件写操作后, 文件读写指针指向文件尾。父进程被唤醒重新上台后, 如果不执行 seek 语句, 则文件读写指针仍然在文件尾, 读操作无法读回 “Hello World! ”。程序输出如下图所示:



```
QEMU
Machine View
welcome to Unix V6++ Tongji's Edition!
[/])#cd bin
[/bin)#fileTest
The file 3 is created.
12 characters have been written into the file 3.
0 characters are read from file 3:
```

(3) 代码 2 中, 子进程先执行 close 操作, 因为此时内存 iNode 和 File 结构还有父进程在使用, 所以只是释放打开文件描述符 fd, 递减 File 结构引用计数 f_count, 发现未减到 0, 则操作结束。父进程后执行 close 操作, 释放打开文件描述符 fd, 递减 File 结构引用计数 f_count, 发现减到 0, 则释放该 File 结构, 递减 iNode 节点引用计数 i_count, 发现减到 0, 则释放该 iNode 节点,

实验 4.3 中的 3 个问题:

代码 3 为满足设计要求的参考代码。

(1) 在代码 3 中, 父子进程共享同一个 iNode 节点, 但是分别有自己的 File 结构, 因而父子进程有不同的读写权限 (父进程为读权限, 子进程为写权限) 和不同的读写指针;

(2) 在这样的共享方式中, 父进程被唤醒重新上台后, 不再需要执行 seek 函数。因为父子进程有不同的读写指针, 子进程对文件的写操作不会影响父进程的读写指针位置。

(3) 代码 3 中, 子进程先执行 close 操作, 释放打开文件描述符 fd, 递减 File 结构引用计数 f_count, 发现减到 0, 则释放该 File 结构, 递减 iNode 节点引用计数 i_count, 发现未减到 0 (父进程的 File 结构还在使用), 则操作结束。父进程后执行 close 操作, 释放打开文件描述符 fd, 递减 File 结构引用计数 f_count, 发现减到 0, 则释放该 File 结构, 递减 iNode 节点引用计数 i_count, 发现减到 0, 则释放该 iNode 节点,

```
#include <stdio.h>
#include <sys.h>
#include <file.h>

void main1()
{
    char data1[13]="Hello World!";
    char data2[13];
    int fd = 0;
    int count = 0;
    int i,j;

    fd = creat("Jessy",0666);    //刚创建好的文件，访问方式是可写
    if (fd>0)
    {
        printf("The file %d is created.\n",fd);
    }
    else
    {
        printf("The file can not be created.\n");
    }
    close(fd);

    if(fork())
    {
        i=wait(&j);
        fd = open("Jessy",1);    //以可读的方式打开文件
        count = read(fd, data2, j);
        printf("%d characters are read from file %d: %s.", count, fd, data2);
        printf("\n");
        close(fd);
    }
    else
    {
        fd = open("Jessy",2);    //以可写的方式打开文件
        count = write(fd, data1, 12);
        if (count == 12)
        {
            printf("%d characters have been written into the file %d.\n", count,fd);
        }
        else
        {
            printf("The file can not be written successfully.\n");
        }
        close(fd);
        exit(count);
    }
}
```

代码 3